# circuits Documentation

***Release 3.2.dev210+ng72e7090.d20151212***

**James Mills**

December 12, 2015

Contents


**1 About** **3**
- 1.1 Examples . . . 3
- 1.2 Features . . . 5
- 1.3 Requirements . . . 5
- 1.4 Supported Platforms . . . 5
- 1.5 Installation . . . 6
- 1.6 License . . . 6
- 1.7 Feedback . . . 6
- 1.8 Community . . . 6

**2 Documentation** **7**
- 2.1 Getting Started . . . 7
- 2.2 circuits Tutorials . . . 9
- 2.3 circuits User Manual . . . 21
- 2.4 circuits.web User Manual . . . 34
- 2.5 API Documentation . . . 50
- 2.6 Developer Docs . . . 291
- 2.7 Change Log . . . 296
- 2.8 Road Map . . . 298
- 2.9 Contributors . . . 298
- 2.10 Frequently Asked Questions . . . 299
- 2.11 Glossary . . . 299
- 2.12 Hello . . . 299
- 2.13 Echo Server . . . 300
- 2.14 Hello Web . . . 300
- 2.15 Documentation TODO . . . 300
- 2.16 PyPi README Page . . . 300

**3 Indices and tables** **305**

**Python Module Index** **307**

**Release** 3.2.dev210+ng72e7090.d20151212

**Date** December 12, 2015

CHAPTER 1

# About

circuits is a **Lightweight Event** driven and **Asynchronous Application Framework** for the Python Programming Language with a strong **Component** Architecture.

circuits also includes a lightweight, high performance and scalable HTTP/WSGI compliant web server as well as various I/O and Networking components.

- Website
- Downloads
- Documentation

Got questions?

- Ask a Question (Tag it: `circuits-framework`)

## 1.1 Examples

### 1.1.1 Hello

```
#!/usr/bin/env python

"""circuits Hello World"""

from circuits import Component, Event


class hello(Event):

    """hello Event"""


class App(Component):

    def hello(self):
        """Hello Event Handler"""

        print("Hello World!")

    def started(self, component):
        """Started Event Handler

```

```
        This is fired internally when your application starts up and can be used to
        trigger events that only occur once during startup.
        """

        self.fire(hello())  # Fire hello Event

        raise SystemExit(0)  # Terminate the Application

App().run()
```

Download Source Code: `hello.py`:

### 1.1.2 Echo Server

```
#!/usr/bin/env python

"""Simple TCP Echo Server

This example shows how you can create a simple TCP Server (an Echo Service)
utilizing the builtin Socket Components that the circuits library ships with.
"""

from circuits import handler, Debugger
from circuits.net.sockets import TCPServer


class EchoServer(TCPServer):

    @handler("read")
    def on_read(self, sock, data):
        """Read Event Handler

        This is fired by the underlying Socket Component when there has been
        new data read from the connected client.

        ..note :: By simply returning, client/server socket components listen
                  to ValueChagned events (feedback) to determine if a handler
                  returned some data and fires a subsequent Write event with
                  the value returned.
        """

        return data

# Start and "run" the system.
# Bind to port 0.0.0.0:8000
app = EchoServer(("0.0.0.0", 8000))
Debugger().register(app)
app.run()
```

Download Source Code: `echoserver.py`:

### 1.1.3 Hello Web

```
#!/usr/bin/env python

```

```
from circuits.web import Server, Controller


class Root(Controller):

    def index(self):
        """Index Request Handler

        Controller(s) expose implicitly methods as request handlers.
        Request Handlers can still be customized by using the ``@expose``
        decorator. For example exposing as a different path.
        """

        return "Hello World!"

app = Server(("0.0.0.0", 8000))
Root().register(app)
app.run()
```

Download Source Code: `helloweb.py`:

More examples...

## 1.2 Features

- event driven
- concurrency support
- component architecture
- asynchronous I/O components
- no required external dependencies
- full featured web framework (circuits.web)
- coroutine based synchronization primitives

## 1.3 Requirements

- circuits has no dependencies beyond the Python Standard Library.

## 1.4 Supported Platforms

- Linux, FreeBSD, Mac OS X, Windows
- Python 2.6, 2.7, 3.2, 3.3, 3.4
- pypy 2.0, 2.1, 2.2

## 1.5 Installation

The simplest and recommended way to install circuits is with pip. You may install the latest stable release from PyPI with pip:

```
$ pip install circuits
```

If you do not have pip, you may use easy_install:

```
$ easy_install circuits
```

Alternatively, you may download the source package from the PyPi or the Downloads extract it and install using:

```
$ python setup.py install
```

**Note:** You can install the development version via `pip install circuits==dev`.

## 1.6 License

circuits is licensed under the MIT License.

## 1.7 Feedback

We welcome any questions or feedback about bugs and suggestions on how to improve circuits.

Let us know what you think about circuits. @pythoncircuits.

Do you have suggestions for improvement? Then please Create an Issue with details of what you would like to see. I'll take a look at it and work with you to either incorporate the idea or find a better solution.

## 1.8 Community

There are also several places you can reach out to the circuits community:

- Mailing List
- Google+ Group
- #circuits IRC Channel on the FreeNode IRC Network
- Ask a Question on Stackoverflow (Tag it: `circuits-framework`)

CHAPTER 2

# Documentation

## 2.1 Getting Started

### 2.1.1 Quick Start Guide

The easiest way to download and install circuits is to use the pip command:

```
$ pip install circuits
```

Now that you have successfully downloaded and installed circuits, let's test that circuits is properly installed and working.

First, let's check the installed version:

```
>>> import circuits
>>> print circuits.__version__
```

This should output:

Try some of the examples in the examples/ directory shipped with the distribution.

Have fun :)

### 2.1.2 Downloading

#### Latest Stable Release

The latest stable releases can be downloaded from the Releases page (*specifically the Tags tab*).

#### Latest Development Source Code

We use Git for source control and code sharing.

The latest development branch can be cloned using the following command:

```
$ git clone https://github.com/circuits/circuits.git
```

For further instructions on how to use Git, please refer to the Git Website.

### 2.1.3 Installing

#### Installing from a Source Package

*If you have downloaded a source archive, this applies to you.*

```
$ python setup.py install
```

For other installation options see:

```
$ python setup.py --help install
```

#### Installing from the Development Repository

*If you have cloned the source code repository, this applies to you.*

If you have cloned the development repository, it is recommended that you use setuptools and use the following command:

```
$ python setup.py develop
```

This will allow you to regularly update your copy of the circuits development repository by simply performing the following in the circuits working directory:

```
$ hg pull -u
```

**Note:** You do not need to reinstall if you have installed with setuptools via the circuits repository and used setuptools to install in "develop" mode.

### 2.1.4 Requirements and Dependencies

- circuits has no **required** dependencies beyond the Python Standard Library.
- Python: >= 2.6 or pypy >= 2.0

  **Supported Platforms** Linux, FreeBSD, Mac OS X, Windows

  **Supported Python Versions** 2.6, 2.7, 3.2, 3.3

  **Supported pypy Versions** 2.0

#### Other Optional Dependencies

These dependencies are not strictly required and only add additional features.

- pydot – For rendering component graphs of an application.
- pyinotify – For asynchronous file system event notifications and the `circuits.io.notify` module.

## 2.2 circuits Tutorials

### 2.2.1 Tutorial

#### Overview

Welcome to the circuits tutorial. This 5-minute tutorial will guide you through the basic concepts of circuits. The goal is to introduce new concepts incrementally with walk-through examples that you can try out! By the time you've finished, you should have a good basic understanding of circuits, how it feels and where to go from there.

#### The Component

First up, let's show how you can use the `Component` and run it in a very simple application.

```
#!/usr/bin/env python

from circuits import Component

Component().run()
```

Download 001.py

Okay so that's pretty boring as it doesn't do very much! But that's okay... Read on!

Let's try to create our own custom Component called `MyComponent`. This is done using normal Python subclassing.

```
#!/usr/bin/env python

from circuits import Component


class MyComponent(Component):

    """My Component"""

MyComponent().run()
```

Download 002.py

Okay, so this still isn't very useful! But at least we can create custom components with the behavior we want.

Let's move on to something more interesting...

---

**Note:** Component(s) in circuits are what sets circuits apart from other Asynchronous or Concurrent Application Frameworks. Components(s) are used as building blocks from simple behaviors to complex ones (*composition of simpler components to form more complex ones*).

---

#### Event Handlers

Let's now extend our little example to say "Hello World!" when it's started.

```
#!/usr/bin/env python

from circuits import Component


```

```
6  class MyComponent(Component):
7
8      def started(self, *args):
9          print("Hello World!")
10
11 MyComponent().run()
```

Download 003.py

Here we've created a simple **Event Handler** that listens for the `started` Event.

---

**Note:** Methods defined in a custom subclassed `Component` are automatically turned into **Event Handlers**. The only exception to this are methods prefixed with an underscore (_).

---

---

**Note:** If you do not want this *automatic* behavior, inherit from `BaseComponent` instead which means you will **have to** use the `~circuits.core.handlers.handler` decorator to define your **Event Handlers**.

---

Running this we get:

```
Hello World!
```

Alright! We have something slightly more useful! Whoohoo it says hello!

---

**Note:** Press ^C (*CTRL* + *C*) to exit.

---

### Registering Components

So now that we've learned how to use a Component, create a custom Component and create simple Event Handlers, let's try something a bit more complex by creating a complex component made up of two simpler ones.

---

**Note:** We call this **Component Composition** which is the very essence of the circuits Application Framework.

---

Let's create two components:

- `Bob`
- `Fred`

```
1  #!/usr/bin/env python
2
3  from circuits import Component
4
5
6  class Bob(Component):
7
8      def started(self, *args):
9          print("Hello I'm Bob!")
10
11
12 class Fred(Component):
13
14     def started(self, *args):
15         print("Hello I'm Fred!")
16
17 (Bob() + Fred()).run()
```

Download 004.py

Notice the way we register the two components Bob and Fred together ? Don't worry if this doesn't make sense right now. Think of it as putting two components together and plugging them into a circuit board.

Running this example produces the following result:

```
Hello I'm Bob!
Hello I'm Fred!
```

Cool! We have two components that each do something and print a simple message on the screen!

### Complex Components

Now, what if we wanted to create a Complex Component? Let's say we wanted to create a new Component made up of two other smaller components?

We can do this by simply registering components to a Complex Component during initialization.

---

**Note:** This is also called **Component Composition** and avoids the classical Diamond problem of Multiple Inheritance. In circuits we do not use Multiple Inheritance to create **Complex Components** made up of two or more base classes of components, we instead compose them together via registration.

---

```
#!/usr/bin/env python

from circuits import Component
from circuits.tools import graph


class Pound(Component):

    def __init__(self):
        super(Pound, self).__init__()

        self.bob = Bob().register(self)
        self.fred = Fred().register(self)

    def started(self, *args):
        print(graph(self.root))


class Bob(Component):

    def started(self, *args):
        print("Hello I'm Bob!")


class Fred(Component):

    def started(self, *args):
        print("Hello I'm Fred!")

Pound().run()
```

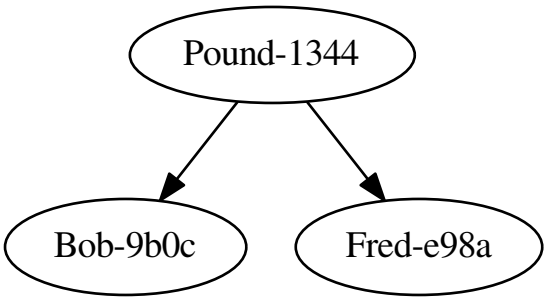
Download 005.py

So now Pound is a Component that consists of two other components registered to it: Bob and Fred

The output of this is identical to the previous:

```
* <Pound/* 3391:MainThread (queued=0, channels=1, handlers=3) [R]>
 * <Bob/* 3391:MainThread (queued=0, channels=1, handlers=1) [S]>
 * <Fred/* 3391:MainThread (queued=0, channels=1, handlers=1) [S]>
Hello I'm Bob!
Hello I'm Fred!
```

The only difference is that `Bob` and `Fred` are now part of a more Complex Component called `Pound`. This can be illustrated by the following diagram:




**Note:** The extra lines in the above output are an ASCII representation of the above graph (*produced by pydot + graphviz*).

Cool :-)

### Component Inheritance

Since circuits is a framework written for the Python Programming Language it naturally inherits properties of Object Orientated Programming (OOP) – such as inheritance.

So let's take our `Bob` and `Fred` components and create a Base Component called `Dog` and modify our two dogs (`Bob` and `Fred`) to subclass this.

```
#!/usr/bin/env python

from circuits import Component, Event


class woof(Event):

    """woof Event"""


class Pound(Component):

    def __init__(self):
        super(Pound, self).__init__()

        self.bob = Bob().register(self)
        self.fred = Fred().register(self)
```

```
18
19      def started(self, *args):
20          self.fire(woof())
21
22
23  class Dog(Component):
24
25      def woof(self):
26          print("Woof! I'm %s!" % self.name)
27
28
29  class Bob(Dog):
30
31      """Bob"""
32
33
34  class Fred(Dog):
35
36      """Fred"""
37
38  Pound().run()
```

Download 006.py

Now let's try to run this and see what happens:

```
Woof! I'm Bob!
Woof! I'm Fred!
```

So both dogs barked! Hmmm

### Component Channels

What if we only want one of our dogs to bark? How do we do this without causing the other one to bark as well?

Easy! Use a separate `channel` like so:

```
1   #!/usr/bin/env python
2
3   from circuits import Component, Event
4
5
6   class woof(Event):
7
8       """woof Event"""
9
10
11  class Pound(Component):
12
13      def __init__(self):
14          super(Pound, self).__init__()
15
16          self.bob = Bob().register(self)
17          self.fred = Fred().register(self)
18
19      def started(self, *args):
20          self.fire(woof(), self.bob)
21
22
```

```
23 class Dog(Component):
24
25     def woof(self):
26         print("Woof! I'm %s!" % self.name)
27
28
29 class Bob(Dog):
30
31     """Bob"""
32
33     channel = "bob"
34
35
36 class Fred(Dog):
37
38     """Fred"""
39
40     channel = "fred"
41
42 Pound().run()
```

Download 007.py

**Note:** Events can be fired with either the `.fire(...)` or `.fireEvent(...)` method.

If you run this, you'll get:

```
Woof! I'm Bob!
```

### Event Objects

So far in our tutorial we have been defining an Event Handler for a builtin Event called `started`. What if we wanted to define our own Event Handlers and our own Events? You've already seen how easy it is to create a new Event Handler by simply defining a normal Python method on a Component.

Defining your own Events helps with documentation and testing and makes things a little easier.

Example:

```
class MyEvent(Event):
    """MyEvent"""
```

So here's our example where we'll define a new Event called `Bark` and make our `Dog` fire a `Bark` event when our application starts up.

```
1  #!/usr/bin/env python
2
3  from circuits import Component, Event
4
5
6  class bark(Event):
7
8      """bark Event"""
9
10
11 class Pound(Component):
12
13     def __init__(self):
```

```
        super(Pound, self).__init__()

        self.bob = Bob().register(self)
        self.fred = Fred().register(self)


class Dog(Component):

    def started(self, *args):
        self.fire(bark())

    def bark(self):
        print("Woof! I'm %s!" % self.name)


class Bob(Dog):

    """Bob"""

    channel = "bob"


class Fred(Dog):

    """Fred"""

    channel = "fred"

Pound().run()
```

Download 008.py

If you run this, you'll get:

```
Woof! I'm Bob!
Woof! I'm Fred!
```

### The Debugger

Lastly...

Asynchronous programming has many advantages but can be a little harder to write and follow. A silently caught exception in an Event Handler, or an Event that never gets fired, or any number of other weird things can cause your application to fail and leave you scratching your head.

Fortunately circuits comes with a `Debugger` Component to help you keep track of what's going on in your application, and allows you to tell what your application is doing.

Let's say that we defined out `bark` Event Handler in our `Dog` Component as follows:

```
def bark(self):
   print("Woof! I'm %s!" % name)
```

Now clearly there is no such variable as `name` in the local scope.

For reference here's the entire example...

```
#!/usr/bin/env python

```

```
3  from circuits import Component, Event
4
5
6  class bark(Event):
7
8      """bark Event"""
9
10
11 class Pound(Component):
12
13     def __init__(self):
14         super(Pound, self).__init__()
15
16         self.bob = Bob().register(self)
17         self.fred = Fred().register(self)
18
19
20 class Dog(Component):
21
22     def started(self, *args):
23         self.fire(bark())
24
25     def bark(self):
26         print("Woof! I'm %s!" % name)   # noqa
27
28
29 class Bob(Dog):
30
31     """Bob"""
32
33     channel = "bob"
34
35
36 class Fred(Dog):
37
38     """Fred"""
39
40     channel = "fred"
41
42 Pound().run()
```

Download 009.py

If you run this, you'll get:

That's right! You get nothing! Why? Well in circuits any error or exception that occurs in a running application is automatically caught and dealt with in a way that lets your application "keep on going". Crashing is unwanted behavior in a system so we expect to be able to recover from horrible situations.

SO what do we do? Well that's easy. circuits comes with a `Debugger` that lets you log all events as well as all errors so you can quickly and easily discover which Event is causing a problem and which Event Handler to look at.

If you change Line 34 of our example...

From:

```
```

To:

```
from circuits import Debugger

(Pound() + Debugger()).run()
```

Then run this, you'll get the following:

```
<Registered[bob:registered] [<Bob/bob 3191:MainThread (queued=0, channels=2, handlers=2) [S]>, <Pound
<Registered[fred:registered] [<Fred/fred 3191:MainThread (queued=0, channels=2, handlers=2) [S]>, <Po
<Registered[*:registered] [<Debugger/* 3191:MainThread (queued=0, channels=1, handlers=1) [S]>, <Pour
<Started[*:started] [<Pound/* 3191:MainThread (queued=0, channels=5, handlers=5) [R]>, None] {}>
<Bark[bob:bark] [] {}>
<Bark[fred:bark] [] {}>
<Error[*:exception] [<type 'exceptions.NameError'>, NameError("global name 'name' is not defined",),
ERROR <listener on ('bark',) {target='bob', priority=0.0}> (<type 'exceptions.NameError'>): global na
  File "/home/prologic/work/circuits/circuits/core/manager.py", line 459, in __handleEvent
 retval = handler(*eargs, **ekwargs)
  File "source/tutorial/009.py", line 22, in bark
    print("Woof! I'm %s!" % name)

<Error[*:exception] [<type 'exceptions.NameError'>, NameError("global name 'name' is not defined",),
ERROR <listener on ('bark',) {target='fred', priority=0.0}> (<type 'exceptions.NameError'>): global n
  File "/home/prologic/work/circuits/circuits/core/manager.py", line 459, in __handleEvent
    retval = handler(*eargs, **ekwargs)
  File "source/tutorial/009.py", line 22, in bark
    print("Woof! I'm %s!" % name)

^C<Signal[*:signal] [2, <frame object at 0x808e8ec>] {}>
<Stopped[*:stopped] [<Pound/* 3191:MainThread (queued=0, channels=5, handlers=5) [S]>] {}>
<Stopped[*:stopped] [<Pound/* 3191:MainThread (queued=0, channels=5, handlers=5) [S]>] {}>
```

You'll notice whereas there was no output before there is now a pretty detailed output with the `Debugger` added to the application. Looking through the output, we find that the application does indeed start correctly, but when we fire our `Bark` Event it coughs up two exceptions, one for each of our dogs (`Bob` and `Fred`).

From the error we can tell where the error is and roughly where to look in the code.

---

**Note:** You'll notice many other events that are displayed in the above output. These are all default events that circuits has builtin which your application can respond to. Each builtin Event has a special meaning with relation to the state of the application at that point.

See: *circuits.core.events* for detailed documentation regarding these events.

---

The correct code for the `bark` Event Handler should be:

```
def bark(self):
    print("Woof! I'm %s!" % self.name)
```

Running again with our correction results in the expected output:

```
Woof! I'm Bob!
Woof! I'm Fred!
```

That's it folks!

Hopefully this gives you a feel of what circuits is all about and an easy tutorial on some of the basic concepts. As you're no doubt itching to get started on your next circuits project, here's some recommended reading:

- `../faq`
- `../api/index`

### 2.2.2 Telnet Tutorial

#### Overview

Welcome to our 2nd circuits tutorial. This tutorial is going to walk you through the telnet Example showing you how to various parts of the circuits component library for building a simple TCP client that also accepts user input.

Be sure you have circuits installed before you start:

```
pip install circuits
```

See: Installing

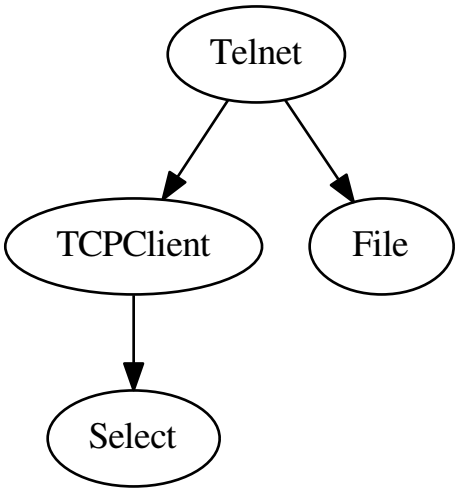
#### Components

You will need the following components:

1. The *TCPClient* Component
2. The *File* Component
3. The *Component* Component

All these are available in the circuits library so there is nothing for you to do. Click on each to read more about them.

#### Design




The above graph is the overall design of our Telnet application. What's shown here is a relationship of how the components fit together and the overall flow of events.

For example:

1. Connect to remote TCP Server.

2. Read input from User.
3. Write input from User to connected Socket.
4. Wait for data from connected Socket and display.

**Note:** The *Select* Component shown is required by our application for Asynchronous I/O polling however we do not need to explicitly use it as it is automatically imported and registered simply by utilizing the *TCPClient* Component.

### Implementation

Without further delay here's the code:

```
#!/usr/bin/env python

import sys


from circuits.io import File
from circuits import handler, Component
from circuits.net.sockets import TCPClient
from circuits.net.events import connect, write


class Telnet(Component):

    channel = "telnet"

    def init(self, host, port):
        self.host = host
        self.port = port

        TCPClient(channel=self.channel).register(self)
        File(sys.stdin, channel="stdin").register(self)

    def ready(self, socket):
        self.fire(connect(self.host, self.port))

    def read(self, data):
        print(data.strip())

    @handler("read", channel="stdin")
    def read_user_input(self, data):
        self.fire(write(data))


host = sys.argv[1]
port = int(sys.argv[2])

Telnet(host, port).run()
```

Download telnet.py

### Discussion

Some important things to note...

1. Notice that we defined a `channel` for out `Telnet` Component?

   This is so that the events of *`TCPClient`* and *`File`* don't collide. Both of these components share a very similar interface in terms of the events they listen to.

```
class Telnet(Component):

    channel = "telnet"
```

2. Notice as well that in defining a `channel` for our `Telnet` Component we've also "registered" the *`TCPClient`* Component so that it has the same channel as our `Telnet` Component.

   Why? We want our `Telnet` Component to receive all of the events of the *`TCPClient`* Component.

```
TCPClient(channel=self.channel).register(self)
```

3. In addition to our *`TCPClient`* Component being registered with the same `channel` as our `Telnet` Component we can also see that we have registered a *`File`* Component however we have chosen a different channel here called `stdin`.

   Why? We don't want the events from *`TCPClient`* and subsequently our `Telnet` Component to collide with the events from *`File`*.

   So we setup a Component for reading user input by using the *`File`* Component and attaching an event handler to our `Telnet` Component but listening to events from our `stdin` channel.

```
File(sys.stdin, channel="stdin").register(self)
```

```
@handler("read", channel="stdin")
def read_user_input(self, data):
    self.fire(write(data))
```

Here is what the event flow would look like if you were to register the *`Debugger`* to the `Telnet` Component.

```
from circuits import Debugger
(Telnet(host, port) + Debugger()).run()
```

```
$ python telnet.py 10.0.0.2 9000
<registered[telnet] (<TCPClient/telnet 21995:MainThread (queued=0) [S]>, <Telnet/telnet 21995:MainThr
<registered[stdin] (<File/stdin 21995:MainThread (queued=0) [S]>, <Telnet/telnet 21995:MainThread (qu
<registered[*] (<Debugger/* 21995:MainThread (queued=0) [S]>, <Telnet/telnet 21995:MainThread (queue
<started[telnet] (<Telnet/telnet 21995:MainThread (queued=4) [R]> )>
<registered[select] (<Select/select 21995:MainThread (queued=0) [S]>, <TCPClient/telnet 21995:MainThr
<ready[telnet] (<TCPClient/telnet 21995:MainThread (queued=0) [S]> )>
<ready[stdin] (<File/stdin 21995:MainThread (queued=0) [S]> )>
<connect[telnet] ('10.0.0.2', 9000 )>
<_open[stdin] ( )>
<connected[telnet] ('10.0.0.2', 9000 )>
<opened[stdin] ('<stdin>', 'r' )>
Hello World!
<_read[stdin] (<open file '<stdin>', mode 'r' at 0x7f32ff5ab0c0> )>
<read[stdin] ('Hello World!\n' )>
<write[telnet] ('Hello World!\n' )>
<_write[telnet] (<socket._socketobject object at 0x11f7f30> )>
<_read[telnet] (<socket._socketobject object at 0x11f7f30> )>
<read[telnet] ('Hello World!\n' )>
Hello World!
^C<signal[telnet] (2, <frame object at 0x12b0a10> )>
<stopped[telnet] (<Telnet/telnet 21995:MainThread (queued=0) [S]> )>
<close[telnet] ( )>
<close[stdin] ( )>
```

```
<disconnected[telnet] ( )>
<closed[stdin] ( )>
```

#### Testing

To try this example out, download a copy of the echoserver Example and copy and paste the full source code of the `Telnet` example above into a file called `telnet.py`.

In one terminal run:

```
$ python echoserver.py
```

In a second terminal run:

```
$ python telnet.py localhost 9000
```

Have fun!

For more examples see examples.

**See also:**

- Frequently Asked Questions
- API Documentation

## 2.3 circuits User Manual

### 2.3.1 Core Library

#### Components

The architectural concept of circuits is to encapsulate system functionality into discrete manageable and reusable units, called *Components*, that interact by sending and handling events that flow throughout the system.

Technically, a circuits *Component* is a Python class that inherits (*directly or indirectly*) from *BaseComponent*.

Components can be sub-classed like any other normal Python class, however components can also be composed of other components and it is natural to do so. These are called *Complex Components*. An example of a Complex Component within the circuits library is the *circuits.web.servers.Server* Component which is comprised of:

- *circuits.net.sockets.TCPServer*
- *circuits.web.servers.BaseServer*
- *circuits.web.http.HTTP*
- *circuits.web.dispatchers.dispatcher.Dispatcher*

---

**Note:** There is no class or other technical means to mark a component as a complex component. Rather, all component instances in a circuits based application belong to some component tree (there may be several), with Complex Components being a subtree within that structure.

---

A Component is attached to the tree by registering with the parent and detached by unregistering itself. See methods:

- *register()*

- *unregister()*

### Component Registration

To register a component use the `register()` method.

```
from circuits import Component


class Foo(Component):
    """Foo Component"""


class  App(Component):
    """App Component"""

    def init(self):
        Foo().register(self)


app = App()
debugger = Debugger().register(app)
app.run()
```

### Unregistering Components

Components are unregistered via the `unregister()` method.

```
debugger.unregister()
```

**Note:** You need a reference to the component you wish to unregister. The `register()` method returns you a reference of the component that was registered.

### Convenient Shorthand Form

After a while when your application becomes rather large and complex with many components and component registrations you will find it cumbersome to type `.register(blah)`.

circuits has several convenient methods for component registration and deregistration that work in an identical fashion to their `register()` and `unregister()` counterparts.

These convenience methods follow normal mathematical operator precedence rules and are implemented by overloading the Python `__add__`, `__iadd__`, `__sub__` and `__isub__`.

The mapping is as follow:

- `register()` map to + and +=
- `unregister()` map to> − and −=

For example the above could have been written as:

```
from circuits import Component


class Foo(Component):
```

```
    """Foo Component"""


class  App(Component):
    """App Component"""

    def init(self):
        self += Foo()


(App() + Debugger()).run()
```

#### Implicit Component Registration(s)

Sometimes it's handy to implicitly register components into another component by simply referencing the other component instance as a class attribute of the other.

Example:

```
>>> from circuits import Component
>>>
>>> class Foo(Component):
...     """Foo Component"""
...
>>> class App(Component):
...     """App Component"""
...
...     foo = Foo()
...
>>> app = App()
>>> app.components
set([<Foo/* 28599:MainThread (queued=0) [S]>])
>>>
```

The telnet Example does this for example.

### Debugger

The *core Debugger* component is the standard way to debug your circuits applications. It services two purposes:

- Logging events as they flow through the system.
- Logging any exceptions that might occurs in your application.

#### Usage

Using the *Debugger* in your application is very straight forward just like any other component in the circuits component library. Simply add it to your application and register it somewhere (*it doesn't matter where*).

Example:

```
from circuits import Component, Debugger


class App(Component):
    """Your Application"""
```

```
app = Appp()
Debugger().register(app)
app.run()
```

#### Sample Output(s)

Here are some example outputs that you should expect to see when using the *Debugger* component in your application.

Example Code:

```
from circuits import Event, Component, Debugger


class foo(Event):
    """foo Event"""


class App(Component):

    def foo(self, x, y):
        return x + y


app = App() + Debugger()
app.start()
```

Run with:

```
python -i app.py
```

Logged Events:

```
<registered[*] (<Debugger/* 27098:App (queued=0) [S]>, <App/* 27098:App (queued=2) [R]> )>
<started[*] (<App/* 27098:App (queued=1) [R]> )>
>>> app.fire(foo(1, 2))
<Value () result=False; errors=False; for <foo[*] (1, 2 )>
>>> <foo[*] (1, 2 )>
```

Logged Exceptions:

```
>>> app.fire(foo())
<Value () result=False; errors=False; for <foo[*] ( )>
>>> <foo[*] ( )>
<exception[*] (<type 'exceptions.TypeError'>, TypeError('foo() takes exactly 3 arguments (1 given)',)
ERROR <handler[*][foo] (App.foo)> (<foo[*] ( )>) {<type 'exceptions.TypeError'>}: foo() takes exactly
  File "/home/prologic/work/circuits/circuits/core/manager.py", line 561, in _dispatcher
    value = handler(*eargs, **ekwargs)
```

## Events

### Basic usage

Events are objects that contain data (*arguments and keyword arguments*) about the message being sent to a receiving component. Events are triggered by using the *fire()* method of any registered component.

Some events in circuits are fired implicitly by the circuits core like the *started* event used in the tutorial or explicitly by components while handling some other event. Once fired, events are dispatched to the components that are interested in these events (*components whose event handlers match events of interest*).

Events are usually fired on one or more channels, allowing components to gather in "interest groups". This is especially useful if you want to reuse basic components such as a *TCPServer*. A *TCPServer* component fires a *read* event for every package of data that it receives. If we did not have support for channels, it would be very difficult to build two servers in a single process without their read events colliding.

Using channels, we can put one server and all components interested in its events on one channel, and another server and the components interested in this other server's events on another channel.

Components are associated with a channel by setting their `channel` class or instance attribute.

**See also:**

*Component*

Besides having a name, events carry additional arbitrary information. This information is passed as arguments or keyword arguments to the constructor. It is then delivered to the event handler method that must have exactly the same number of arguments and keyword arguments. Of course, as is usual in Python, you can also pass additional information by setting attributes of the event object, though this usage pattern is discouraged.

#### Filtering

Events can be filtered by stopping other event handlers from continuing to process the event.

To do this, simply call the *stop()* method.

Example:

```
@handler("foo")
def stop_foo(self, event, *args, **kwargs):
    event.stop()
```

Here any other event handlers also listening to "foo" will not be processed.

---

**Note:** It's important to use priority event handlers here in this case as all event handlers and events run with the same priority unless explicitly told otherwise.

---

Changed in version 3.0: In circuits 2.x you declared your event handler to be a filter by using `@handler(filter=True)` and returned a `True`-ish value from the respective event handler to achieve the same effect. This is **no longer** the case in circuits 3.x Please use `event.stop()` as noted above.

#### Events as result collectors

Apart from delivering information to handlers, event objects may also collect information. If a handler returns something that is not `None`, it is stored in the event's `value` attribute. If a second (or any subsequent) handler invocation also returns a value, the values are stored as a list. Note that the value attribute is of type *Value* and you must access its property `value` to access the data stored (`collected_information = event.value.value`).

The collected information can be accessed by handlers in order to find out about any return values from the previously invoked handlers. More useful though, is the possibility to access the information after all handlers have been invoked. After all handlers have run successfully (i.e. no handler has thrown an error) circuits may generate an event that indicates the successful handling. This event has the name of the event just handled with "Success" appended. So if the event is called `Identify` then the success event is called `IdentifySuccess`. Success events aren't delivered

by default. If you want successful handling to be indicated for an event, you have to set the optional attribute `success` of this event to `True`.

The handler for a success event must be defined with two arguments. When invoked, the first argument is the event just having been handled successfully and the second argument is (as a convenience) what has been collected in `event.value.value` (note that the first argument may not be called `event`, for an explanation of this restriction as well as for an explanation why the method is called `identify_success` see the section on handlers).

```
#!/usr/bin/env python

from circuits import Component, Debugger, Event


class Identify(Event):

    """Identify Event"""

    success = True


class Pound(Component):

    def __init__(self):
        super(Pound, self).__init__()

        Debugger().register(self)
        Bob().register(self)
        Fred().register(self)

    def started(self, *args):
        self.fire(Identify())

    def Identify_success(self, evt, result):
        if not isinstance(result, list):
            result = [result]
        print "In pound:"
        for name in result:
            print name


class Dog(Component):

    def Identify(self):
        return self.__class__.__name__


class Bob(Dog):

    """Bob"""


class Fred(Dog):

    """Fred"""

Pound().run()
```

Download handler_returns.py

#### Advanced usage

Sometimes it may be necessary to take some action when all state changes triggered by an event are in effect. In this case it is not sufficient to wait for the completion of all handlers for this particular event. Rather, we also have to wait until all events that have been fired by those handlers have been processed (and again wait for the events fired by those events' handlers, and so on). To support this scenario, circuits can fire a `Complete` event. The usage is similar to the previously described success event. Details can be found in the API description of *`circuits.core.events.Event`*.

### Handlers

#### Explicit Event Handlers

Event Handlers are methods of components that are invoked when a matching event is dispatched. These can be declared explicitly on a *`BaseComponent`* or *`Component`* or by using the *`handler()`* decorator.

```
1  #!/usr/bin/env python
2
3  from circuits import handler, BaseComponent, Debugger
4
5
6  class MyComponent(BaseComponent):
7
8      def __init__(self):
9          super(MyComponent, self).__init__()
10
11         Debugger().register(self)
12
13     @handler("started", channel="*")
14     def system_started(self, component):
15         print "Start event detected"
16
17 MyComponent().run()
```

Download handler_annotation.py

The handler decorator on line 14 turned the method `system_started` into an event handler for the event `started`.

When defining explicit event handlers in this way, it's convention to use the following pattern:

```
@handler("foo")
def print_foobar(self, ...):
    print("FooBar!")
```

This makes reading code clear and concise and obvious to the reader that the method is not part of the class's public API (*leading underscore as per Python convention*) and that it is invoked for events of type `SomeEvent`.

The optional keyword argument "`channel`" can be used to attach the handler to a different channel than the component's channel (*as specified by the component's channel attribute*).

Handler methods must be declared with arguments and keyword arguments that match the arguments passed to the event upon its creation. Looking at the API for *`started`* you'll find that the component that has been started is passed as an argument to its constructor. Therefore, our handler method must declare one argument (*Line 14*).

The *`handler()`* decorator accepts other keyword arguments that influence the behavior of the event handler and its invocation. Details can be found in the API description of *`handler()`*.

### Implicit Event Handlers

To make things easier for the developer when creating many event handlers and thus save on some typing, the *Component* can be used and subclassed instead which provides an implicit mechanism for creating event handlers.

Basically every method in the component is automatically and implicitly marked as an event handler with `@handler(<name>)` where `<name>` is the name of each method applied.

The only exceptions are:

- Methods that start with an underscore _.
- Methods already marked explicitly with the *handler()* decorator.

Example:

```
#!/usr/bin/env python


from circuits import handler, Component, Event


class hello(Event):
    """hello Event"""


class App(Component):

    def _say(self, message):
        """Print the given message

        This is a private method as denoted via the prefixed underscore.
        This will not be turned into an event handler.
        """

        print(message)

    def started(self, manager):
        self._say("App Started!")
        self.fire(hello())
        raise SystemExit(0)

    @handler("hello")
    def print_hello(self):
        """hello Event Handlers

        Print "Hello World!" when the ``hello`` Event is received.

        As this is already decorated with the ``@handler``
        decorator, it will be left as it is and won't get
        touched by the implicit event handler creation
        mechanisms.
        """

        print("Hello World!")

    @handler(False)
    def test(self, *args, **kwargs):
        """A simple test method that does nothing
```

```
        This will not be turned into an event handlers
        because of the ``False`` argument passed to the
        ``@handler`` decorator. This only makes sense
        when subclassing ``Component`` and you want to
        have fine grained control over what methods
        are not turned into event handlers.
        """

        pass

App().run()
```

**Note:** You can specify that a method will not be marked as an event handler by passing `False` as the first argument to `@handler()`.

### Manager

The *core Manager* class is the base class of all components in circuits. It is what defines the API(s) of all components and necessary machinery to run your application smoothly.

**Note:** It is not recommended to actually use the *Manager* in your application code unless you know what you're doing.

**Warning:** A *Manager* **does not** know how to register itself to other components! It is a manager, not a component, however it does form the basis of every component.

#### Usage

Using the *Manager* in your application is not really recommended except in some special circumstances where you want to have a top-level object that you can register things to.

Example:

```
from circuits import Component, Manager


class App(Component):
    """Your Application"""


manager = Manager()
App().register(manager)
manager.run()
```

**Note:** If you *think* you need a *Manager* chances are you probably don't. Use a *Component* instead.

### Values

The *core Value* class is an internal part of circuits' Futures and Promises used to fulfill promises of the return value of an event handler and any associated chains of events and event handlers.

Basically when you fire an event `foo()` such as:

```
x = self.fire(foo())
```

`x` here is an instance of the *Value* class which will contain the value returned by the event handler for `foo` in the `.value` property.

---

**Note:** There is also *getValue()* which can be used to also retrieve the underlying value held in the instance of the *Value* class but you should not need to use this as the `.value` property takes care of this for you.

---

The only other API you may need in your application is the `notify` which can be used to trigger a `value_changed` event when the underlying *Value* of the event handler has changed. In this way you can do something asynchronously with the event handler's return value no matter when it finishes.

Example Code:

```
#!/usr/bin/python -i


from circuits import handler, Event, Component, Debugger


class hello(Event):
    "hello Event"


class test(Event):
    "test Event"


class App(Component):

    def hello(self):
        return "Hello World!"

    def test(self):
        return self.fire(hello())

    @handler("hello_value_changed")
    def _on_hello_value_changed(self, value):
        print("hello's return value was: {}".format(value))


app = App()
Debugger().register(app)
```

Example Session:

```
$ python -i ../app.py
>>> x = app.fire(test())
>>> x.notify = True
>>> app.tick()
<registered[*] (<Debugger/* 27798:MainThread (queued=0) [S]>, <App/* 27798:MainThread (queued=1) [S]>
<test[*] ( )>
>>> app.tick()
<hello[*] ( )>
>>> app.tick()
<test_value_changed[<App/* 27798:MainThread (queued=0) [S]>] (<Value ('Hello World!') result=True; e
>>> app.tick()
```

```
>>> x
<Value ('Hello World!') result=True; errors=False; for <test[*] ( )>
>>> x.value
'Hello World!'
>>>
```

The `Value.notify` attribute can also be set to the name of an event which should be used to fire the `value_changed` event to.

If the form `x.notify = True` used then the event that gets fired is a concatenation of the original event and the `value_changed` event. e.g: `foo_value_changed`.

---

**Note:** This is a bit advanced and should only be used by experienced users of the circuits framework. If you simply want basic synchronization of event handlers it's recommended that you try the `circuits.Component.call()` and `circuits.Component.wait()` synchronization primitives first.

---

## 2.3.2 Miscellaneous

### Tools

There are two main tools of interest in circuits. These are:

- *circuits.tools.inspect()*
- *circuits.tools.graph()*

These can be found in the *circuits.tools* module.

#### Introspecting your Application

The *inspect()* function is used to help introspect your application by displaying all the channels and events handlers defined through the system including any additional meta data about them.

Example:

```
>>> from circuits import Component
>>> class App(Component):
...     def foo(self):
...             pass
...
>>> app = App()
>>> from circuits.tools import inspect
>>> print(inspect(app))
 Components: 0

 Event Handlers: 3
  unregister; 1
   <handler[*][unregister] (App._on_unregister)>
  foo; 1
   <handler[*][foo] (App.foo)>
  prepare_unregister_complete; 1
   <handler[<instance of App>][prepare_unregister_complete] (App._on_prepare_unregister_complete)>
```

### Displaying a Visual Representation of your Application

The *graph()* function is used to help visualize the different components in your application and how they interact with one another and how they are registered in the system.

In order to get a image from this you must have the following packages installed:

- networkx
- pygraphviz
- matplotlib

You can install the required dependencies via:

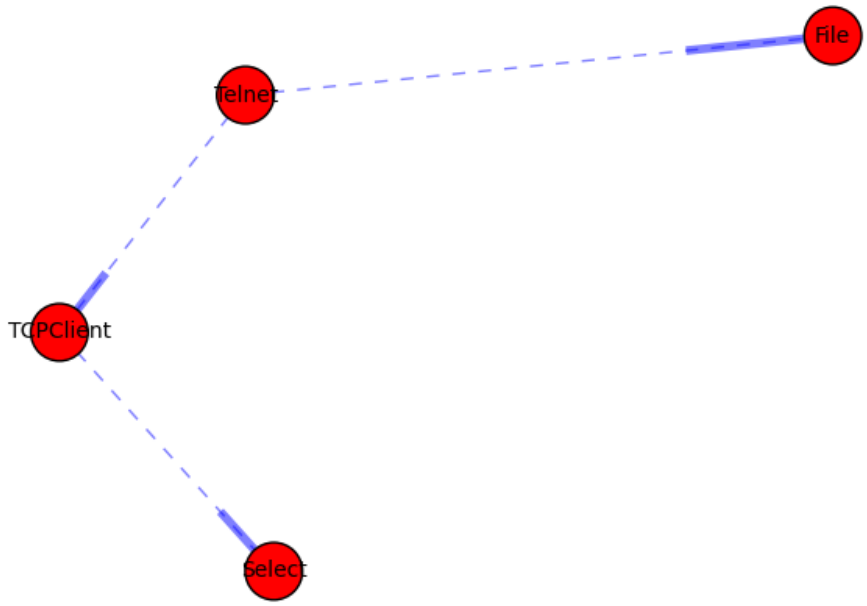
```
pip install matplotlib networkx pygraphviz
```

Example:

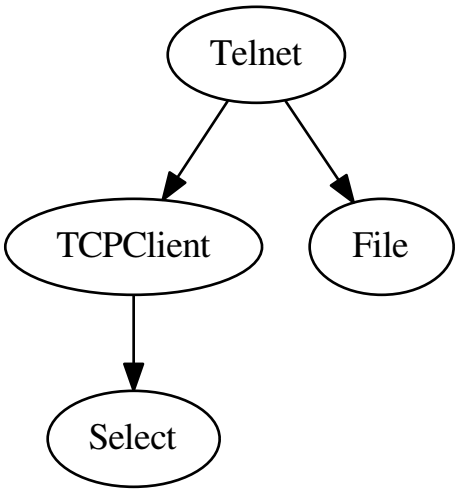
```
>>> from circuits import Component, Debugger
>>> from circuits.net.events import write
>>> from circuits.net.sockets import TCPServer
>>>
>>> class EchoServer(Component):
...     def init(self, host="0.0.0.0", port=8000):
...             TCPServer((host, port)).register(self)
...             Debugger().register(self)
...     def read(self, sock, data):
...             self.fire(write(sock, data))
...
>>> server = EchoServer()
>>>
>>> from circuits.tools import graph
>>> print(graph(server))
* <EchoServer/* 784:MainThread (queued=2) [S]>
 * <TCPServer/server 784:MainThread (queued=0) [S]>
 * <Debugger/* 784:MainThread (queued=0) [S]>
```

An output image will be saved to your current working directory and by called `<name>.png` where **<name>** is the name of the top-level component in your application of the value you pass to the `name=` keyword argument of `~circuits.tools.graph`.

Example output of telnet Example:




And its DOT Graph:



## 2.4 circuits.web User Manual

### 2.4.1 Introduction

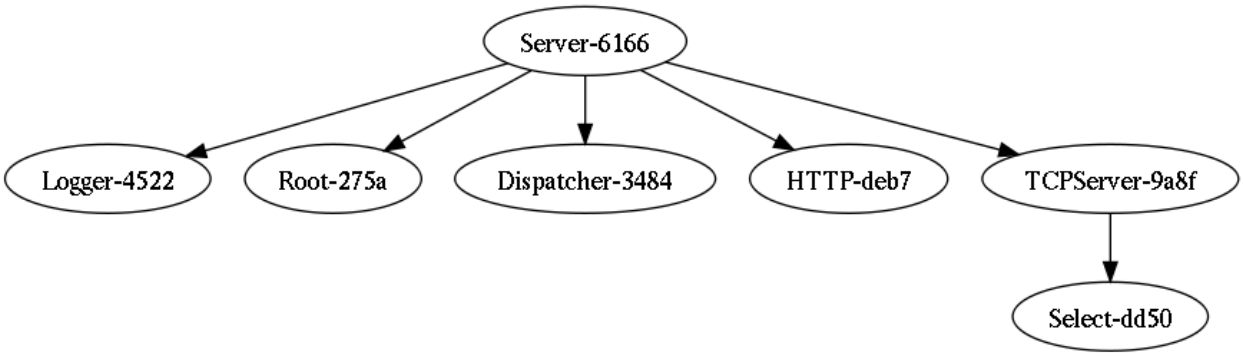
circuits.web is a set of components for building high performance HTTP/1.1 and WSGI/1.0 compliant web applications. These components make it easy to rapidly develop rich, scalable web applications with minimal effort.

circuits.web borrows from

- CherryPy
- BaseHTTPServer (*Python std. lib*)
- wsgiref (*Python std. lib*)

### 2.4.2 Getting Started

Just like any application or system built with circuits, a circuits.web application follows the standard Component based design and structure whereby functionality is encapsulated in components. circuits.web itself is designed and built in this fashion. For example a circuits.web Server's structure looks like this:




To illustrate the basic steps, we will demonstrate developing your classical "Hello World!" applications in a web-based way with circuits.web

To get started, we first import the necessary components:

```
from circutis.web import Server, Controller
```

Next we define our first Controller with a single Request Handler defined as our index. We simply return "Hello World!" as the response for our Request Handler.

```
class Root(Controller):

   def index(self):
      return "Hello World!"
```

This completes our simple web application which will respond with "Hello World!" when anyone accesses it.

*Admittedly this is a stupidly simple web application! But circuits.web is very powerful and plays nice with other tools.*

Now we need to run the application:

```
(Server(8000) + Root()).run()
```

That's it! Navigate to: http://127.0.0.1:8000/ and see the result.

Here's the complete code:

```
from circuits.web import Server, Controller

class Root(Controller):

   def index(self):
      return "Hello World!"

(Server(8000) + Root()).run()
```

Have fun!

### 2.4.3 Features

circuits.web is not a **Full Stack** or **High Level** web framework, rather it is more closely aligned with CherryPy and offers enough functionality to make quickly developing web applications easy and as flexible as possible.

circuits.web does not provide high level features such as:

- Templating
- Database access
- Form Validation
- Model View Controller
- Object Relational Mapper

The functionality that circutis.web **does** provide ensures that circuits.web is fully HTTP/1.1 and WSGI/1.0 compliant and offers all the essential tools you need to build your web application or website.

To demonstrate each feature, we're going to use the classical "Hello World!" example as demonstrated earlier in Getting Started.

Here's the code again for easy reference:

```
from circuits.web import Server, Controller


class Root(Controller):

    def index(self):
        return "Hello World!"


(Server(8000) + Root()).run()
```

#### Logging

circuits.web's *Logger* component allows you to add logging support compatible with Apache log file formats to your web application.

To use the `Logger` simply add it to your application:

```
(Server(8000) + Logger() + Root()).run()
```

Example Log Output:

```
127.0.0.1 - - [05/Apr/2014:10:13:01] "GET / HTTP/1.1" 200 12 "" "curl/7.35.0"
127.0.0.1 - - [05/Apr/2014:10:13:02] "GET /docs/build/html/index.html HTTP/1.1" 200 22402 "" "curl/7.
```

### Cookies

Access to cookies are provided through the *Request* Object which holds data about the request. The attribute `cookie` is provided as part of the *Request* Object. It is a dict-like object, an instance of `Cookie.SimpleCookie` from the python standard library.

To demonstrate "Using Cookies" we'll write a very simple application that remembers who we are:

If a cookie **name** is found, display "Hello <name>!". Otherwise, display "Hello World!" If an argument is given or a query parameter **name** is given, store this as the **name** for the cookie. Here's how we do it:

```
from circuits.web import Server, Controller


class Root(Controller):

    def index(self, name=None):
        if name:
            self.cookie["name"] = name
        else:
            name = self.cookie.get("name", None)
            name = "World!" if name is None else name.value

        return "Hello {0:s}!".format(name)


(Server(8000) + Root()).run()
```

**Note:** To access the actual value of a cookie use the `.value` attribute.

**Warning:** Cookies can be vulnerable to XSS (*Cross Site Scripting*) attacks so use them at your own risk. See: http://en.wikipedia.org/wiki/Cross-site_scripting#Cookie_security

### Dispatchers

circuits.web provides several dispatchers in the *dispatchers* module. Most of these are available directly from the circuits.web namespace by simply importing the required "dispatcher" from circuits.web.

Example:

```
from circuits.web import Static
```

The most important "dispatcher" is the default *Dispatcher* used by the circuits.web *Server* to dispatch incoming requests onto a channel mapping (*remember that circuits is event-driven and uses channels*), quite similar to that of CherryPy or any other web framework that supports object traversal.

Normally you don't have to worry about any of the details of the *default* *Dispatcher* nor do you have to import it or use it in any way as it's already included as part of the circuits.web *Server* Component structure.

### Static

The *Static* "dispatcher" is used for serving static resources/files in your application. To use this, simply add it to your application. It takes some optional configuration which affects it's behavior.

The simplest example (*as per our Base Example*):

```
(Server(8000) + Static() + Root()).run()
```

This will serve up files in the *current directory* as static resources.

---

**Note:** This may override your **index** request handler of your top-most (Root) *Controller*. As this might be undesirable and it's normally common to serve static resources via a different path and even have them stored in a separate physical file path, you can configure the Static "dispatcher".

---

Static files stored in /home/joe/www/:

```
(Server(8000) + Static(docroot="/home/joe/www/") + Root()).run()
```

Static files stored in /home/joe/www/ **and** we want them served up as /static URI(s):

```
(Server(8000) + Static("/static", docroot="/home/joe/www/") + Root()).run()
```

### Dispatcher

The *Dispatcher* (*the default*) is used to dispatch requests and map them onto channels with a similar URL Mapping as CherryPy's. A set of "paths" are maintained by the Dispatcher as Controller(s) are registered to the system or unregistered from it. A channel mapping is found by traversing the set of known paths (*Controller(s)*) and successively matching parts of the path (*split by /*) until a suitable Controller and Request Handler is found. If no Request Handler is found that matches but there is a "default" Request Handler, it is used.

This Dispatcher also included support for matching against HTTP methods:

- GET
- POST
- PUT
- DELETE.

Here are some examples:

```
class Root(Controller):

    def index(self):
        return "Hello World!"

    def foo(self, arg1, arg2, arg3):
        return "Foo: %r, %r, %r" % (arg1, arg2, arg3)

    def bar(self, kwarg1="foo", kwarg2="bar"):
        return "Bar: kwarg1=%r, kwarg2=%r" % (kwarg1, kwarg2)

    def foobar(self, arg1, kwarg1="foo"):
        return "FooBar: %r, kwarg1=%r" % (arg1, kwarg1)
```

With the following requests:

```
http://127.0.0.1:8000/
http://127.0.0.1:8000/foo/1/2/3
http://127.0.0.1:8000/bar?kwarg1=1
http://127.0.0.1:8000/bar?kwarg1=1&kwarg=2
http://127.0.0.1:8000/foobar/1
http://127.0.0.1:8000/foobar/1?kwarg1=1
```

The following output is produced:

```
Hello World!
Foo: '1', '2', '3'
Bar: kwargs1='1', kwargs2='bar'
Bar: kwargs1='1', kwargs2='bar'
FooBar: '1', kwargs1='foo'
FooBar: '1', kwargs1='1'
```

This demonstrates how the Dispatcher handles basic paths and how it handles extra parts of a path as well as the query string. These are essentially translated into arguments and keyword arguments.

To define a Request Handler that is specifically for the HTTP `POST` method, simply define a Request Handler like:

```
class Root(Controller):

    def index(self):
        return "Hello World!"


class Test(Controller):

    channel = "/test"

    def POST(self, *args, **kwargs): #***
        return "%r %r" % (args, kwargs)
```

This will handles `POST` requests to "/test", which brings us to the final point of creating URL structures in your application. As seen above to create a sub-structure of Request Handlers (*a tree*) simply create another *`Controller`* Component giving it a different channel and add it to the system along with your existing Controller(s).

**Warning:** All public methods defined in your `If you don't want something exposed either subclass from :class:`~BaseController` whereby you have to explicitly use `expose()` or use `@expose(False)` to decorate a public method as **NOT Exposed** or simply prefix the desired method with an underscore (e.g: `def _foo(...):`).

### VirtualHosts

The *`VirtualHosts`* "dispatcher" allows you to serves up different parts of your application for different "virtual" hosts.

Consider for example you have the following hosts defined:

```
localdomain
foo.localdomain
bar.localdomain
```

You want to display something different on the default domain name "localdomain" and something different for each of the sub-domains "foo.localdomain" and "bar.localdomain".

To do this, we use the VirtualHosts "dispatcher":

```
from circuits.web import Server, Controller, VirtualHosts


class Root(Controller):

    def index(self):
        return "I am the main vhost"


class Foo(Controller):

    channel = "/foo"

    def index(self):
        return "I am foo."


class Bar(Controller):

    channel = "/bar"

    def index(self):
        return "I am bar."


domains = {
    "foo.localdomain:8000": "foo",
    "bar.localdomain:8000": "bar",
}


(Server(8000) + VirtualHosts(domains) + Root() + Foo() + Bar()).run()
```

With the following requests:

```
http://localdomain:8000/
http://foo.localdomain:8000/
http://bar.localdomain:8000/
```

The following output is produced:

```
I am the main vhost
I am foo.
I am bar.
```

The argument **domains** pasted to VirtualHosts' constructor is a mapping (*dict*) of: domain -> channel

#### XMLRPC

The *XMLRPC* "dispatcher" provides a circuits.web application with the capability of serving up RPC Requests encoded in XML (XML-RPC).

Without going into too much details (*if you're using any kind of RPC "dispatcher" you should know what you're doing...*), here is a simple example:

```
from circuits import Component
from circuits.web import Server, Logger, XMLRPC

```

```
4
5  class Test(Component):
6
7      def foo(self, a, b, c):
8          return a, b, c
9
10
11 (Server(8000) + Logger() + XMLRPC() + Test()).run()
```

Here is a simple interactive session:

```
>>> import xmlrpclib
>>> xmlrpc = xmlrpclib.ServerProxy("http://127.0.0.1:8000/rpc/")
>>> xmlrpc.foo(1, 2, 3)
[1, 2, 3]
>>>
```

### JSONRPC

The *JSONRPC* "dispatcher" is Identical in functionality to the *XMLRPC* "dispatcher".

Example:

```
1  from circuits import Component
2  from circuits.web import Server, Logger, JSONRPC
3
4
5  class Test(Component):
6
7      def foo(self, a, b, c):
8          return a, b, c
9
10
11 (Server(8000) + Logger() + JSONRPC() + Test()).run()
```

Interactive session (*requires the 'jsonrpclib <https://pypi.python.org/pypi/jsonrpc>'_ library*):

```
>>> import jsonrpclib
>>> jsonrpc = jsonrpclib.ServerProxy("http://127.0.0.1:8000/rpc/")
>>> jsonrpc.foo(1, 2, 3)
{'result': [1, 2, 3], 'version': '1.1', 'id': 2, 'error': None}
>>>
```

## Caching

circuits.web includes all the usual **Cache Control**, **Expires** and **ETag** caching mechanisms.

For simple expires style caching use the `expires()` tool from *circuits.web.tools*.

Example:

```
1  from circuits.web import Server, Controller
2
3
4  class Root(Controller):
5
6      def index(self):
7          self.expires(3600)
```

```
        return "Hello World!"


(Server(8000) + Root()).run()
```

For other caching mechanisms and validation please refer to the *circuits.web.tools* documentation.

See in particular:

- `expires()`
- `validate_since()`

---

**Note:** In the example above we used `self.expires(3600)` which is just a convenience method built into the *Controller*. The *Controller* has other such convenience methods such as `.uri`, `.forbidden()`, `.redirect()`, `.notfound()`, `.serve_file()`, `.serve_download()` and `.expires()`.

These are just wrappers around *tools* and *events*.

---

### Compression

circuits.web includes the necessary low-level tools in order to achieve compression. These tools are provided as a set of functions that can be applied to the response before it is sent to the client.

Here's how you can create a simple Component that enables compression in your web application or website.

```
from circuits import handler, Component

from circuits.web.tools import gzip
from circuits.web import Server, Controller, Logger


class Gzip(Component):

    @handler("response", priority=1.0)
    def compress_response(self, event, response):
        event[0] = gzip(response)


class Root(Controller):

    def index(self):
        return "Hello World!"


(Server(8000) + Gzip() + Root()).run()
```

Please refer to the documentation for further details:

- *tools.gzip()*
- *utils.compress()*

### Authentication

circuits.web provides both HTTP Plain and Digest Authentication provided by the functions in *circuits.web.tools*:

- *tools.basic_auth()*
- *tools.check_auth()*
- *tools.digest_auth()*

The first 2 arguments are always (*as with most circuits.web tools*):

- `(request, response)`

An example demonstrating the use of "Basic Auth":

```
from circuits.web import Server, Controller
from circuits.web.tools import check_auth, basic_auth


class Root(Controller):

    def index(self):
        realm = "Test"
        users = {"admin": "admin"}
        encrypt = str

        if check_auth(self.request, self.response, realm, users, encrypt):
            return "Hello %s" % self.request.login

        return basic_auth(self.request, self.response, realm, users, encrypt)


(Server(8000) + Root()).run()
```

For "Digest Auth":

```
from circuits.web import Server, Controller
from circuits.web.tools import check_auth, digest_auth


class Root(Controller):

    def index(self):
        realm = "Test"
        users = {"admin": "admin"}
        encrypt = str

        if check_auth(self.request, self.response, realm, users, encrypt):
            return "Hello %s" % self.request.login

        return digest_auth(self.request, self.response, realm, users, encrypt)


(Server(8000) + Root()).run()
```

### Session Handling

Session Handling in circuits.web is very similar to Cookies. A dict-like object called **.session** is attached to every Request Object during the life-cycle of that request. Internally a Cookie named **circuits.session** is set in the response.

Rewriting the Cookie Example to use a session instead:

```
from circuits.web import Server, Controller, Sessions


class Root(Controller):

    def index(self, name=None):
        if name:
            self.session["name"] = name
        else:
            name = self.session.get("name", "World!")

        return "Hello %s!" % name


(Server(8000) + Sessions() + Root()).run()
```

**Note:** The only Session Handling provided is a temporary in-memory based one and will not persist. No future Session Handling components are planned. For persistent data you should use some kind of Database.

### 2.4.4 How To Guides

These "How To" guides will steer you in the right direction for common aspects of modern web applications and website design.

#### How Do I: Use a Templating Engine

circuits.web tries to stay out of your way as much as possible and doesn't impose any restrictions on what external libraries and tools you can use throughout your web application or website. As such you can use any template language/engine you wish.

##### Example: Using Mako

This basic example of using the Mako Templating Language. First a TemplateLookup instance is created. Finally a function called `render(name, **d)` is created that is used by Request Handlers to render a given template and apply data to it.

Here is the basic example:

```
#!/usr/bin/env python

import os


import mako
from mako.lookup import TemplateLookup


from circuits.web import Server, Controller


templates = TemplateLookup(
    directories=[os.path.join(os.path.dirname(__file__), "tpl")],
    module_directory="/tmp",
```

```
    output_encoding="utf-8"
)


def render(name, **d): #**
    try:
        return templates.get_template(name).render(**d) #**
    except:
        return mako.exceptions.html_error_template().render()


class Root(Controller):

    def index(self):
        return render("index.html")

    def submit(self, firstName, lastName):
        msg = "Thank you %s %s" % (firstName, lastName)
        return render("index.html", message=msg)


(Server(8000) + Root()).run()
```

#### Other Examples

Other Templating engines will be quite similar to integrate.

### How Do I: Integrate with a Database

> **Warning:** Using databases in an asynchronous framework is problematic because most database implementations don't support asynchronous I/O operations.
> Generally the solution is to use threading to hand off database operations to a separate thread.

Here are some ways to help integrate databases into your application:

1. Ensure your queries are optimized and do not block for extensive periods of time.
2. Use a library like SQLAlchemy that supports multi-threaded database operations to help prevent your circuits.web web application from blocking.
3. *Optionally* take advantage of the `Worker` component to fire `task` events wrapping database calls in a thread or process pool. You can then use the `call()` and `wait()` synchronization primitives to help with the control flow of your requests and responses.

Another way you can help improve performance is by load balancing across multiple backends of your web application. Using things like haproxy or nginx for load balancing can really help.

### How Do I: Use WebSockets

Since the `WebSocketDispatcher` id a circuits.web "dispatcher" it's quite easy to integrate into your web application. Here's a simple trivial example:

```
#!/usr/bin/env python

from circuits.net.events import write
from circuits import Component, Debugger
from circuits.web.dispatchers import WebSockets
from circuits.web import Controller, Logger, Server, Static


class Echo(Component):

    channel = "wsserver"

    def read(self, sock, data):
        self.fireEvent(write(sock, "Received: " + data))


class Root(Controller):

    def index(self):
        return "Hello World!"


app = Server(("0.0.0.0", 8000))
Debugger().register(app)
Static().register(app)
Echo().register(app)
Root().register(app)
Logger().register(app)
WebSockets("/websocket").register(app)
app.run()
```

See the circuits.web examples.

### How do I: Build a Simple Form

circuits.web parses all POST data as a request comes through and creates a dictionary of kwargs (Keyword Arguments) that are passed to Request Handlers.

Here is a simple example of handling form data:

```
#!/usr/bin/env python

from circuits.web import Server, Controller


class Root(Controller):

    html = """\
<html>
 <head>
  <title>Basic Form Handling</title>
 </head>
 <body>
  <h1>Basic Form Handling</h1>
  <p>
   Example of using
   <a href="http://code.google.com/p/circuits/">circuits</a> and it's
   <b>Web Components</b> to build a simple web application that handles
```

```
   some basic form data.
  </p>
  <form action="submit" method="POST">
   <table border="0" rules="none">
    <tr>
     <td>First Name:</td>
     <td><input type="text" name="firstName"></td>
    </tr>
    <tr>
     <td>Last Name:</td>
     <td><input type="text" name="lastName"></td>
    </tr>
     <tr>
      <td colspan=2" align="center">
       <input type="submit" value="Submit">
     </td>
     </tr>
   </table>
  </form>
 </body>
</html>"""


    def index(self):
        return self.html

    def submit(self, firstName, lastName):
        return "Hello %s %s" % (firstName, lastName)


(Server(8000) + Root()).run(
```

### How Do I: Upload a File

You can easily handle File Uploads as well using the same techniques as above. Basically the "name" you give your <input> tag of type="file" will get passed as the Keyword Argument to your Request Handler. It has the following two attributes:

```
.filename - The name of the uploaded file.
.value - The contents of the uploaded file.
```

Here's the code!

```
#!/usr/bin/env python

from circuits.web import Server, Controller


UPLOAD_FORM = """
<html>
 <head>
  <title>Upload Form</title>
 </head>
 <body>
  <h1>Upload Form</h1>
  <form method="POST" action="/" enctype="multipart/form-data">
   Description: <input type="text" name="desc"><br>
```

```
    <input type="file" name="file">
    <input type="submit" value="Submit">
  </form>
 </body>
</html>
"""

UPLOADED_FILE = """
<html>
 <head>
  <title>Uploaded File</title>
 </head>
 <body>
  <h1>Uploaded File</h1>
  <p>
   Filename: %s<br>
   Description: %s
  </p>
  <p><b>File Contents:</b></p>
  <pre>
  %s
  </pre>
 </body>
</html>
"""


class Root(Controller):

    def index(self, file=None, desc=""):
        if file is None:
            return UPLOAD_FORM
        else:
            filename = file.filename
            return UPLOADED_FILE % (file.filename, desc, file.value)


(Server(8000) + Root()).run()
```

circuits.web automatically handles form and file uploads and gives you access to the uploaded file via arguments to the request handler after they've been processed by the dispatcher.

### How Do I: Integrate with WSGI Applications

Integrating with other WSGI Applications is quite easy to do. Simply add in an instance of the *Gateway* component into your circuits.web application.

Example:

```
#!/usr/bin/env python

from circuits.web.wsgi import Gateway
from circuits.web import Controller, Server


def foo(environ, start_response):
    start_response("200 OK", [("Content-Type", "text/plain")])
    return ["Foo!"]
```

```
10
11
12 class Root(Controller):
13     """App Rot"""
14
15     def index(self):
16         return "Hello World!"
17
18
19 app = Server(("0.0.0.0", 10000))
20 Root().register(app)
21 Gateway({"/foo": foo}).register(app)
22 app.run()
```

The `apps` argument of the *`Gateway`* component takes a key/value pair of `path -> callable` (*a Python dictionary*) that maps each URI to a given WSGI callable.

### How Do I: Deploy with Apache and mod_wsgi

Here's how to deploy your new Circuits powered Web Application on Apache using mod_wsgi.

Let's say you have a Web Hosting account with some provider.

- Your Username is: "joblogs"
- Your URL is: http://example.com/~joeblogs/
- Your Docroot is: /home/joeblogs/www/

#### Configuring Apache

The first step is to add in the following .htaccess file to tell Apache hat we want any and all requests to http://example.com/~joeblogs/ to be served up by our circuits.web application.

Created the .htaccess file in your **Docroot**:

```
ReWriteEngine On
ReWriteCond %{REQUEST_FILENAME} !-f
ReWriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ /~joeblogs/index.wsgi/$1 [QSA,PT,L]
```

#### Running your Application with Apache/mod_wsgi

The get your Web Application working and deployed on Apache using mod_wsgi, you need to make a few changes to your code. Based on our Basic Hello World example earlier, we modify it to the following:

```
1  #!/usr/bin/env python
2
3  from circuits.web import Controller
4  from circuits.web.wsgi import Application
5
6
7  class Root(Controller):
8
9      def index(self):
10         return "Hello World!"
```

```
11
12
13 application = Application() + Root()
```

That's it! To run this, save it as index.wsgi and place it in your Web Root (public-html or www directory) as per the above guidelines and point your favorite Web Browser to: http://example.com/~joeblogs/

---

**Note:** It is recommended that you actually use a reverse proxy setup for deploying circuits.web web application so that you don't loose the advantages and functionality of using an event-driven component architecture in your web apps.

In **production** you should use a load balance and reverse proxy combination for best performance.

---

### 2.4.5 Miscellaneous

#### Writing Tools

Most of the internal tools used by circuits.web in circuits.web.tools are simply functions that modify the Request or Response objects in some way or another... We won't be covering that here... What we will cover is how to build simple tools that do something to the Request or Response along it's life-cycle.

Here is a simple example of a tool that uses the pytidylib library to tidy up the HTML output before it gets sent back to the requesting client.

Code:

```
1  #!/usr/bin/env python
2  from tidylib import tidy_document
3
4  from circuits import Component
5
6  class Tidy(Component):
7
8      channel = "http"
9
10     def response(self, response):
11         document, errors = tidy_document("".join(response.body))
12         response.body = document
13 Usage:
14
15 (Server(8000) + Tidy() + Root()).run()
```

**How it works:**

This tool works by intercepting the Response Event on the "response" channel of the "http" target (*or Component*). For more information about the life cycle of Request and Response events, their channels and where and how they can be intercepted to perform various tasks read the Request/Response Life Cycle section.

#### Writing Dispatchers

In circuits.web writing a custom "dispatcher" is only a matter of writing a Component that listens for incoming Request events on the "request" channel of the "web" target. The simplest kind of "dispatcher" is one that simply modifies the request.path in some way. To demonstrate this we'll illustrate and describe how the !VirtualHosts "dispatcher" works.

VirtualHosts code:

```
class VirtualHosts(Component):

    channel = "web"

    def __init__(self, domains):
        super(VirtualHosts, self).__init__()

        self.domains = domains

    @handler("request", filter=True, priority=1)
    def request(self, event, request, response):
        path = request.path.strip("/")

        header = request.headers.get
        domain = header("X-Forwarded-Host", header("Host", ""))
        prefix = self.domains.get(domain, "")

        if prefix:
            path = _urljoin("/%s/" % prefix, path)
            request.path = path
```

The important thing here to note is the Event Handler listening on the appropriate channel and the request.path being modified appropriately.

You'll also note that in [source:circuits/web/dispatchers.py] all of the dispatchers have a set priority. These priorities are defined as:

```
$ grin "priority" circuits/web/dispatchers/
circuits/web/dispatchers/dispatcher.py:
   92 :     @handler("request", filter=True, priority=0.1)
circuits/web/dispatchers/jsonrpc.py:
   38 :     @handler("request", filter=True, priority=0.2)
circuits/web/dispatchers/static.py:
   59 :     @handler("request", filter=True, priority=0.9)
circuits/web/dispatchers/virtualhosts.py:
   49 :     @handler("request", filter=True, priority=1.0)
circuits/web/dispatchers/websockets.py:
   53 :     @handler("request", filter=True, priority=0.2)
circuits/web/dispatchers/xmlrpc.py:
   36 :     @handler("request", filter=True, priority=0.2)
```

in web applications that use multiple dispatchers these priorities set precedences for each "dispatcher" over another in terms of who's handling the Request Event before the other.

---

**Note:** Some dispatchers are designed to filter the Request Event and prevent it from being processed by other dispatchers in the system.

---

## 2.5 API Documentation

### 2.5.1 circuits package

#### Subpackages

##### circuits.app package

### Submodules

**circuits.app.daemon module** Daemon Component

Component to daemonize a system into the background and detach it from its controlling PTY. Supports PID file writing, logging stdin, stdout and stderr and changing the current working directory.

**class** circuits.app.daemon.**Daemon**(**args*, ***kwargs*)
Bases: *circuits.core.components.Component*

Daemon Component

**Parameters**

- **pidfile** (*str or unicode*) – .pid filename
- **stdin** (*str or unicode*) – filename to log stdin
- **stdout** (*str or unicode*) – filename to log stdout
- **stderr** (*str or unicode*) – filename to log stderr

initializes x; see x.__class__.__doc__ for signature

**channel = 'daemon'**

**daemonize**()

**deletepid**()

**init**(*pidfile*, *path='/'*, *stdin=None*, *stdout=None*, *stderr=None*, *channel='daemon'*)

**on_started**(*component*)

**registered**(*component*, *manager*)

**writepid**()

**class** circuits.app.daemon.**daemonize**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

daemonize Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'daemonize'**

**class** `circuits.app.daemon.`**deletepid**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

"deletepid Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'deletepid'**

**class** circuits.app.daemon.**writepid**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

"writepid Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'writepid'**

**Module contents** Application Components

Contains various components useful for application development and tasks common to applications.

**class** circuits.app.**Daemon**(**args*, ***kwargs*)
Bases: *circuits.core.components.Component*

Daemon Component

**Parameters**

- **pidfile** (*str or unicode*) – .pid filename
- **stdin** (*str or unicode*) – filename to log stdin
- **stdout** (*str or unicode*) – filename to log stdout
- **stderr** (*str or unicode*) – filename to log stderr

initializes x; see x.__class__.__doc__ for signature

**channel = 'daemon'**

**daemonize**()

**deletepid**()

**init**(*pidfile*, *path='/'*, *stdin=None*, *stdout=None*, *stderr=None*, *channel='daemon'*)

**on_started**(*component*)

**registered**(*component*, *manager*)

**writepid**()

**class** circuits.app.**DropPrivileges**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**drop_privileges**()

**init**(*user='nobody'*, *group='nobody'*, *umask=63*, ***kwargs*)

**on_ready**(*server*, *bind*)

### circuits.core package

#### Submodules

#### circuits.core.bridge module Bridge

The Bridge Component is used for inter-process communications between processes. Bridge is used internally when a Component is started in "process mode" via circuits.core.manager.start(). Typically a Pipe is used as the socket transport between two sides of a Bridge (*there must be a :class:'~Bridge' instnace on both sides*).

**class** circuits.core.bridge.**ipc**(*event*, *channel=None*)
Bases: *circuits.core.events.Event*

ipc Event

Send an event to a child/parent process

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **channel** (*str*) – IPC Channel (channel to use on child/parent).

**name = 'ipc'**

**class** circuits.core.bridge.**Bridge**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'bridge'**

**init**(*socket*, *channel='bridge'*)

**circuits.core.components module** This module defines the BaseComponent and its subclass Component.

**class** circuits.core.components.**prepare_unregister**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

This event is fired when a component is about to be unregistered from the component tree. Unregistering a component actually detaches the complete subtree that the unregistered component is the root of. Components that need to know if they are removed from the main tree (e.g. because they maintain relationships to other components in the tree) handle this event, check if the component being unregistered is one of their ancestors and act accordingly.

**Parameters** **component** – the component that will be unregistered

**complete = True**

**in_subtree**(*component*)
Convenience method that checks if the given *component* is in the subtree that is about to be detached.

**name = 'prepare_unregister'**

**class** circuits.core.components.**BaseComponent**(**args*, ***kwargs*)
Bases: *circuits.core.manager.Manager*

This is the base class for all components in a circuits based application. Components can (and should, except for root components) be registered with a parent component.

BaseComponents can declare methods as event handlers using the handler decoration (see *circuits.core.handlers.handler()*). The handlers are invoked for matching events from the component's channel (specified as the component's `channel` attribute).

BaseComponents inherit from *circuits.core.manager.Manager*. This provides components with the *circuits.core.manager.Manager.fireEvent()* method that can be used to fire events as the result of some computation.

Apart from the `fireEvent()` method, the Manager nature is important for root components that are started or run.

**Variables** ***channel*** – a component can be associated with a specific channel by setting this attribute. This should either be done by specifying a class attribute *channel* in the derived class or by passing a keyword parameter *channel="..."* to *__init__*. If specified, the component's handlers receive events on the specified channel only, and events fired by the component will be sent on the specified channel (this behavior may be overridden, see *Event*, *fireEvent()* and *handler()*). By default, the channel attribute is set to "*", meaning that events are fired on all channels and received from all channels.

initializes x; see x.__class__.__doc__ for signature

**channel = '*'**

**register**(*parent*)
Inserts this component in the component tree as a child of the given *parent* node.

**Parameters** **parent** (*Manager*) – the parent component after registration has completed.

This method fires a `Registered` event to inform other components in the tree about the new member.

**unregister**()
Removes this component from the component tree.

Removing a component from the component tree is a two stage process. First, the component is marked as to be removed, which prevents it from receiving further events, and a *prepare_unregister* event is fired. This allows other components to e.g. release references to the component to be removed before it is actually removed from the component tree.

After the processing of the `prepare_unregister` event has completed, the component is removed from the tree and an *`unregistered`* event is fired.

**unregister_pending**

**classmethod handlers**()
Returns a list of all event handlers for this Component

**classmethod events**()
Returns a list of all events this Component listens to

**classmethod handles**(**names*)
Returns True if all names are event handlers of this Component

**class** circuits.core.components.**Component**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If you use Component instead of BaseComponent as base class for your own component class, then all methods that are not marked as private (i.e: start with an underscore) are automatically decorated as handlers.

The methods are invoked for all events from the component's channel where the event's name matches the method's name.

**circuits.core.debugger module** Debugger component used to debug each event in a system by printing each event to sys.stderr or to a Logger Component instance.

**class** circuits.core.debugger.**Debugger**(*errors=True*, *events=True*, *file=None*, *logger=None*, *prefix=None*, *trim=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Create a new Debugger Component

Creates a new Debugger Component that listens to all events in the system printing each event to sys.stderr or a Logger Component.

**Variables**

- ***IgnoreEvents*** – list of events (str) to ignore
- ***IgnoreChannels*** – list of channels (str) to ignore
- **enabled** – Enabled/Disabled flag

**Parameters** **log** – Logger Component instance or None (*default*)

initializes x; see x.__class__.__doc__ for signature

**IgnoreEvents = ['generate_events']**

**IgnoreChannels = []**

**circuits.core.events module** This module defines the basic event class and common events.

**class** circuits.core.events.**EventType**
Bases: type

**class** circuits.core.events.**Event**(**args*, ***kwargs*)
Bases: object

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**channels = ()**
The channels this message is sent to.

**parent = None**

**notify = False**

**success = False**

**failure = False**

**complete = False**

**alert_done = False**

**waitingHandlers = 0**

**classmethod create**(*name*, **args*, ***kwargs*)

**child**(*name*, **args*, ***kwargs*)

**name = 'Event'**

**cancel**()
Cancel the event from being processed (if not already)

**stop**()
Stop further processing of this event

**class** circuits.core.events.**exception**(*type*, *value*, *traceback*, *handler=None*, *fevent=None*)
Bases: *circuits.core.events.Event*

exception Event

This event is sent for any exceptions that occur during the execution of an event Handler that is not SystemExit or KeyboardInterrupt.

**Parameters**

- **type** (*type*) – type of exception
- **value** (*exceptions.TypeError*) – exception object
- **traceback** (traceback) – traceback of exception
- **handler** (*@handler(<method>)*) – handler that raised the exception
- **fevent** (*event*) – event that failed

**name = 'exception'**

**class** circuits.core.events.**started**(*manager*)
Bases: *circuits.core.events.Event*

started Event

This Event is sent when a Component or Manager has started running.

**Parameters** **manager** (*Component or Manager*) – The component or manager that was started

**name = 'started'**

**class** circuits.core.events.**stopped**(*manager*)
Bases: *circuits.core.events.Event*

stopped Event

This Event is sent when a Component or Manager has stopped running.

**Parameters** **manager** (*Component or Manager*) – The component or manager that has stopped

**name = 'stopped'**

**class** circuits.core.events.**signal**(*signo*, *stack*)
Bases: *circuits.core.events.Event*

signal Event

This Event is sent when a Component receives a signal.

**Parameters**

- **signo** – The signal number received.
- **stack** – The interrupted stack frame.

**name = 'signal'**

**class** circuits.core.events.**registered**(*component*, *manager*)
Bases: *circuits.core.events.Event*

registered Event

This Event is sent when a Component has registered with another Component or Manager. This Event is only sent if the Component or Manager being registered which is not itself.

**Parameters**

- **component** (Component) – The Component being registered
- **manager** (*Component or Manager*) – The Component or Manager being registered with

**name = 'registered'**

**class** circuits.core.events.**unregistered**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

unregistered Event

This Event is sent when a Component has been unregistered from its Component or Manager.

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'unregistered'**

**class** circuits.core.events.**generate_events**(*lock*, *max_wait*)
Bases: *circuits.core.events.Event*

generate_events Event

This Event is sent by the circuits core. All components that generate timed events or events from external sources (e.g. data becoming available) should fire any pending events in their "generate_events" handler.

The handler must either call `stop()` (*preventing other handlers from being called in the same iteration) or must invoke *reduce_time_left()* with parameter 0.

**Parameters** **max_wait** – maximum time available for generating events.

Components that actually consume time waiting for events to be generated, thus suspending normal execution, must provide a method `resume` that interrupts waiting for events.

**time_left**
The time left for generating events. A value less than 0 indicates unlimited time. You should have only one component in your system (usually a poller component) that spends up to "time left" until it generates an event.

**reduce_time_left**(*time_left*)
Update the time left for generating events. This is typically used by event generators that currently don't want to generate an event but know that they will within a certain time. By reducing the time left, they make sure that they are reinvoked when the time for generating the event has come (at the latest).

This method can only be used to reduce the time left. If the parameter is larger than the current value of time left, it is ignored.

If the time left is reduced to 0 and the event is currently being handled, the handler's *resume* method is invoked.

**lock**

**name = 'generate_events'**

**circuits.core.handlers module** This module define the @handler decorator/function and the HandlesType type.

circuits.core.handlers.**handler**(**names*, ***kwargs*)
Creates an Event Handler

This decorator can be applied to methods of classes derived from *circuits.core.components.BaseComponent*. It marks the method as a handler for the events passed as arguments to the `@handler` decorator. The events are specified by their name.

The decorated method's arguments must match the arguments passed to the *circuits.core.events.Event* on creation. Optionally, the method may have an additional first argument named *event*. If declared, the event object that caused the handler to be invoked is assigned to it.

By default, the handler is invoked by the component's root *Manager* for events that are propagated on the channel determined by the BaseComponent's *channel* attribute. This may be overridden by specifying a different channel as a keyword parameter of the decorator (`channel=...`).

Keyword argument `priority` influences the order in which handlers for a specific event are invoked. The higher the priority, the earlier the handler is executed.

If you want to override a handler defined in a base class of your component, you must specify `override=True`, else your method becomes an additional handler for the event.

**Return value**

Normally, the results returned by the handlers for an event are simply collected in the *circuits.core.events.Event*'s `value` attribute. As a special case, a handler may return a `types.GeneratorType`. This signals to the dispatcher that the handler isn't ready to deliver a result yet. Rather, it has interrupted it's execution with a `yield None` statement, thus preserving its current execution state.

The dispatcher saves the returned generator object as a task. All tasks are reexamined (i.e. their `next()` method is invoked) when the pending events have been executed.

This feature avoids an unnecessarily complicated chaining of event handlers. Imagine a handler A that needs the results from firing an event E in order to complete. Then without this feature, the final action of A would be to fire event E, and another handler for an event `SuccessE` would be required to complete handler A's operation, now having the result from invoking E available (actually it's even a bit more complicated).

Using this "suspend" feature, the handler simply fires event E and then yields `None` until e.g. it finds a result in E's `value` attribute. For the simplest scenario, there even is a utility method *circuits.core.manager.Manager.callEvent()* that combines firing and waiting.

**class** circuits.core.handlers.**Unknown**
Bases: object

Unknown Dummy Component

circuits.core.handlers.**reprhandler**(*handler*)

**class** circuits.core.handlers.**HandlerMetaClass**(*name*, *bases*, *ns*)
Bases: type

**circuits.core.helpers module**

**class** circuits.core.helpers.**FallBackGenerator**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**resume**()
Implements the resume method as required from components that handle `GenerateEvents`.

**class** circuits.core.helpers.**FallBackExceptionHandler**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If there is no handler for error events in the component hierarchy, this component's handler is added automatically. It simply prints the error information on stderr.

initializes x; see x.__class__.__doc__ for signature

**class** circuits.core.helpers.**FallBackSignalHandler**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If there is no handler for signal events in the component hierarchy, this component's handler is added automatically. It simply terminates the system if the signal is SIGINT or SIGTERM.

initializes x; see x.__class__.__doc__ for signature

**circuits.core.loader module** This module implements a generic Loader suitable for dynamically loading components from other modules. This supports loading from local paths, eggs and zip archives. Both setuptools and distribute are fully supported.

**class** circuits.core.loader.**Loader**(*auto_register=True*, *init_args=None*, *init_kwargs=None*, *paths=None*, *channel='loader'*)
Bases: *circuits.core.components.BaseComponent*

Create a new Loader Component

Creates a new Loader Component that enables dynamic loading of components from modules either in local paths, eggs or zip archives.

initializes x; see x.__class__.__doc__ for signature

**channel = 'loader'**

**load**(*name*)

**circuits.core.manager module** This module defines the Manager class.

**exception** circuits.core.manager.**UnregistrableError**
Bases: exceptions.Exception

Raised if a component cannot be registered as child.

**exception** circuits.core.manager.**TimeoutError**
Bases: exceptions.Exception

Raised if wait event timeout occurred

**class** circuits.core.manager.**CallValue**(*value*)
Bases: object

**class** circuits.core.manager.**ExceptionWrapper**(*exception*)
Bases: object

**extract**()

**class** circuits.core.manager.**Sleep**(*seconds*)
Bases: *circuits.six.Iterator*

**expired**

**task**

circuits.core.manager.**sleep**(*seconds*)
Delay execution of a coroutine for a given number of seconds. The argument may be a floating point number for subsecond precision.

**class** circuits.core.manager.**Manager**(**args*, ***kwargs*)
Bases: object

The manager class has two roles. As a base class for component implementation, it provides methods for event and handler management. The method *fireEvent()* appends a new event at the end of the event queue for later execution. *waitEvent()* suspends the execution of a handler until all handlers for a given event have been invoked. *callEvent()* combines the last two methods in a single method.

The methods *addHandler()* and *removeHandler()* allow handlers for events to be added and removed dynamically. (The more common way to register a handler is to use the *handler()* decorator or derive the class from *Component*.)

In its second role, the *Manager* takes the role of the event executor. Every component hierarchy has a root component that maintains a queue of events. Firing an event effectively means appending it to the event queue maintained by the root manager. The *flush()* method removes all pending events from the queue and, for each event, invokes all the handlers. Usually, *flush()* is indirectly invoked by *run()*.

The manager optionally provides information about the execution of events as automatically generated events. If an *Event* has its success attribute set to True, the manager fires a Success event if all handlers have been executed without error. Note that this event will be enqueued (and dispatched) immediately after the events that have been fired by the event's handlers. So the success event indicates both the successful invocation of all handlers for the event and the processing of the immediate follow-up events fired by those handlers.

Sometimes it is not sufficient to know that an event and its immediate follow-up events have been processed. Rather, it is important to know when all state changes triggered by an event, directly or indirectly, have been performed. This also includes the processing of events that have been fired when invoking the handlers for the follow-up events and the processing of events that have again been fired by those handlers and so on. The completion of the processing of an event and all its direct or indirect follow-up events may be indicated by a Complete event. This event is generated by the manager if *Event* has its complete attribute set to True.

Apart from the event queue, the root manager also maintains a list of tasks, actually Python generators, that are updated when the event queue has been flushed.

initializes x; see x.__class__.__doc__ for signature

**name**
Return the name of this Component/Manager

**running**
Return the running state of this Component/Manager

**pid**
Return the process id of this Component/Manager

**getHandlers**(*event*, *channel*, ***kwargs*)

**addHandler**(*f*)

**removeHandler**(*method*, *event=None*)

**registerChild**(*component*)

**unregisterChild**(*component*)

**fireEvent**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**fire**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**registerTask**(*g*)

**unregisterTask**(*g*)

**waitEvent**(*event*, **channels*, ***kwargs*)

**wait**(*event*, **channels*, ***kwargs*)

**callEvent**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**call**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**flushEvents**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**flush**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**start**(*process=False*, *link=None*)
Start a new thread or process that invokes this manager's `run()` method. The invocation of this method returns immediately after the task or process has been started.

**join**()

**stop**(*code=None*)
Stop this manager. Invoking this method causes an invocation of `run()` to return.

**processTask**(*event*, *task*, *parent=None*)

**tick**(*timeout=-1*)
Execute all possible actions once. Process all registered tasks and flush the event queue. If the application is running fire a GenerateEvents to get new events from sources.

This method is usually invoked from *run()*. It may also be used to build an application specific main loop.

**Parameters** **timeout** (*float, measuring seconds*) – the maximum waiting time spent in this method. If negative, the method may block until at least one action has been taken.

**run**(*socket=None*)
Run this manager. The method fires the `Started` event and then continuously calls *tick()*.

The method returns when the manager's *stop()* method is invoked.

If invoked by a programs main thread, a signal handler for the `INT` and `TERM` signals is installed. This handler fires the corresponding `Signal` events and then calls *stop()* for the manager.

**circuits.core.pollers module** Poller Components for asynchronous file and socket I/O.

This module contains Poller components that enable polling of file or socket descriptors for read/write events. Pollers: - Select - Poll - EPoll

**class** circuits.core.pollers.**BasePoller**(*channel=None*)
Bases: *circuits.core.components.BaseComponent*

**channel = None**

**resume**()

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**isReading**(*fd*)

**isWriting**(*fd*)

**discard**(*fd*)

**getTarget**(*fd*)

**class** circuits.core.pollers.**Select**(...) → new Select Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new Select Poller Component that uses the select poller implementation. This poller is not recommended but is available for legacy reasons as most systems implement select-based polling for backwards compatibility.

**channel = 'select'**

**class** circuits.core.pollers.**Poll**(...) → new Poll Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new Poll Poller Component that uses the poll poller implementation.

**channel = 'poll'**

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**discard**(*fd*)

**class** circuits.core.pollers.**EPoll**(...) → new EPoll Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new EPoll Poller Component that uses the epoll poller implementation.

**channel = 'epoll'**

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**discard**(*fd*)

**class** circuits.core.pollers.**KQueue**(...) → new KQueue Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new KQueue Poller Component that uses the kqueue poller implementation.

**channel = 'kqueue'**

**addReader**(*source*, *sock*)

**addWriter**(*source*, *sock*)

**removeReader**(*sock*)

**removeWriter**(*sock*)

**discard**(*sock*)

circuits.core.pollers.**Poller**
alias of *Select*

**circuits.core.timers module** Timer component to facilitate timed events.

**class** circuits.core.timers.**Timer**(*interval*, *event*, **channels*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Timer Component

A timer is a component that fires an event once after a certain delay or periodically at a regular interval.

**Parameters**

- **interval** (`datetime` or number of seconds as a `float`) – the delay or interval to wait for until the event is fired. If interval is specified as datetime, the interval is recalculated as the time span from now to the given datetime.
- **event** (*Event*) – the event to fire.
- **persist** (`bool`) – An optional keyword argument which if `True` will cause the event to be fired repeatedly once per configured interval until the timer is unregistered. **Default:** `False`

**reset**(*interval=None*)
Reset the timer, i.e. clear the amount of time already waited for.

**expiry**

**circuits.core.utils module** Utils

This module defines utilities used by circuits.

`circuits.core.utils.`**flatten**(*root*, *visited=None*)

`circuits.core.utils.`**findchannel**(*root*, *channel*, *all=False*)

`circuits.core.utils.`**findtype**(*root*, *component*, *all=False*)

`circuits.core.utils.`**findcmp**(*root*, *component*, *all=False*)

`circuits.core.utils.`**findroot**(*component*)

`circuits.core.utils.`**safeimport**(*name*)

**circuits.core.values module** This defines the Value object used by components and events.

**class** `circuits.core.values.`**Value**(*event=None*, *manager=None*)
Bases: `object`

Create a new future Value Object

Creates a new future Value Object which is used by Event Objects and the Manager to store the result(s) of an Event Handler's exeuction of some Event in the system.

**Parameters**

- **event** (*Event instance*) – The Event this Value is associated with.
- **manager** (*A Manager/Component instance.*) – The Manager/Component used to trigger notifications.

**Variables**

- **result** – True if this value has been changed.
- ***errors*** – True if while setting this value an exception occured.
- ***notify*** – True or an event name to notify of changes to this value

This is a Future/Promise implementation.

**inform**(*force=False*)

**getValue**(*recursive=True*)

**setValue**(*value*)

**value**
Value of this Value

**circuits.core.workers module** Workers

Worker components used to perform "work" in independent threads or processes. Worker(s) are typically used by a Pool (circuits.core.pools) to create a pool of workers. Worker(s) are not registered with a Manager or another Component - instead they are managed by the Pool. If a Worker is used independently it should not be registered as it causes its main event handler `_on_task` to execute in the other thread blocking it.

**class** `circuits.core.workers.`**`task`**(*f*, **args*, ***kwargs*)
Bases: *circuits.core.events.Event*

task Event

This Event is used to initiate a new task to be performed by a Worker or a Pool of Worker(s).

**Parameters**

- **`f`** (*function*) – The function to be executed.
- **`args`** (*tuple*) – Arguments to pass to the function
- **`kwargs`** (*dict*) – Keyword Arguments to pass to the function

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**`success`** = True

**`failure`** = True

**`name`** = 'task'

**class** `circuits.core.workers.`**`Worker`**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

A thread/process Worker Component

This Component creates a Worker (either a thread or process) which when given a `Task`, will execute the given function in the task in the background in its thread/process.

**Parameters** **`process`** (*bool*) – True to start this Worker as a process (Thread otherwise)

initializes x; see x.__class__.__doc__ for signature

**`channel`** = 'worker'

**`init`**(*process=False*, *workers=None*, *channel='worker'*)

**Module contents** Core

This package contains the essential core parts of the circuits framework.

`circuits.core.`**`handler`**(**names*, ***kwargs*)
Creates an Event Handler

This decorator can be applied to methods of classes derived from *circuits.core.components.BaseComponent*. It marks the method as a handler for the events passed as arguments to the `@handler` decorator. The events are specified by their name.

The decorated method's arguments must match the arguments passed to the *circuits.core.events.Event* on creation. Optionally, the method may have an additional first argument named *event*. If declared, the event object that caused the handler to be invoked is assigned to it.

By default, the handler is invoked by the component's root *Manager* for events that are propagated on the channel determined by the BaseComponent's *channel* attribute. This may be overridden by specifying a different channel as a keyword parameter of the decorator (`channel=...`).

Keyword argument `priority` influences the order in which handlers for a specific event are invoked. The higher the priority, the earlier the handler is executed.

If you want to override a handler defined in a base class of your component, you must specify `override=True`, else your method becomes an additional handler for the event.

**Return value**

Normally, the results returned by the handlers for an event are simply collected in the *circuits.core.events.Event*'s `value` attribute. As a special case, a handler may return a `types.GeneratorType`. This signals to the dispatcher that the handler isn't ready to deliver a result yet. Rather, it has interrupted it's execution with a `yield None` statement, thus preserving its current execution state.

The dispatcher saves the returned generator object as a task. All tasks are reexamined (i.e. their `next()` method is invoked) when the pending events have been executed.

This feature avoids an unnecessarily complicated chaining of event handlers. Imagine a handler A that needs the results from firing an event E in order to complete. Then without this feature, the final action of A would be to fire event E, and another handler for an event `SuccessE` would be required to complete handler A's operation, now having the result from invoking E available (actually it's even a bit more complicated).

Using this "suspend" feature, the handler simply fires event E and then yields `None` until e.g. it finds a result in E's `value` attribute. For the simplest scenario, there even is a utility method *circuits.core.manager.Manager.callEvent()* that combines firing and waiting.

**class** `circuits.core.`**`BaseComponent`**(**args*, ***kwargs*)
Bases: *circuits.core.manager.Manager*

This is the base class for all components in a circuits based application. Components can (and should, except for root components) be registered with a parent component.

BaseComponents can declare methods as event handlers using the handler decoration (see *circuits.core.handlers.handler()*). The handlers are invoked for matching events from the component's channel (specified as the component's `channel` attribute).

BaseComponents inherit from *circuits.core.manager.Manager*. This provides components with the *circuits.core.manager.Manager.fireEvent()* method that can be used to fire events as the result of some computation.

Apart from the `fireEvent()` method, the Manager nature is important for root components that are started or run.

**Variables** ***channel*** – a component can be associated with a specific channel by setting this attribute. This should either be done by specifying a class attribute *channel* in the derived class or by passing a keyword parameter *channel="..."* to *__init__*. If specified, the component's handlers receive events on the specified channel only, and events fired by the component will be sent on the specified channel (this behavior may be overridden, see *Event*, *fireEvent()* and *handler()*). By default, the channel attribute is set to "*", meaning that events are fired on all channels and received from all channels.

initializes x; see x.__class__.__doc__ for signature

**`channel`** = '*'

**classmethod `events`**()
Returns a list of all events this Component listens to

**classmethod handlers**()
Returns a list of all event handlers for this Component

**classmethod handles**(**names*)
Returns True if all names are event handlers of this Component

**register**(*parent*)
Inserts this component in the component tree as a child of the given *parent* node.

**Parameters** **parent** (*Manager*) – the parent component after registration has completed.

This method fires a Registered event to inform other components in the tree about the new member.

**unregister**()
Removes this component from the component tree.

Removing a component from the component tree is a two stage process. First, the component is marked as to be removed, which prevents it from receiving further events, and a *prepare_unregister* event is fired. This allows other components to e.g. release references to the component to be removed before it is actually removed from the component tree.

After the processing of the prepare_unregister event has completed, the component is removed from the tree and an *unregistered* event is fired.

**unregister_pending**

**class** circuits.core.**Component**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**class** circuits.core.**Event**(**args*, ***kwargs*)
Bases: object

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to True, an associated event success (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**alert_done = False**

**cancel**()
Cancel the event from being processed (if not already)

**channels = ()**

**child**(*name*, **args*, ***kwargs*)

**complete = False**

**classmethod create**(*name*, **args*, ***kwargs*)

**failure = False**

**name = 'Event'**

**notify = False**

**parent = None**

**stop**()
Stop further processing of this event

**success = False**

**waitingHandlers = 0**

**class** circuits.core.**task**(*f*, **args*, ***kwargs*)
Bases: *circuits.core.events.Event*

task Event

This Event is used to initiate a new task to be performed by a Worker or a Pool of Worker(s).

**Parameters**

- **f** (*function*) – The function to be executed.
- **args** (*tuple*) – Arguments to pass to the function
- **kwargs** (*dict*) – Keyword Arguments to pass to the function

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**failure = True**

**name = 'task'**

**success = True**

**class** circuits.core.**Worker**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

A thread/process Worker Component

This Component creates a Worker (either a thread or process) which when given a `Task`, will execute the given function in the task in the background in its thread/process.

**Parameters** **process** (*bool*) – True to start this Worker as a process (Thread otherwise)

initializes x; see x.__class__.__doc__ for signature

**channel = 'worker'**

**init** (*process=False*, *workers=None*, *channel='worker'*)

**class** circuits.core.**ipc** (*event*, *channel=None*)
Bases: *circuits.core.events.Event*

ipc Event

Send an event to a child/parent process

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **channel** (*str*) – IPC Channel (channel to use on child/parent).

**name = 'ipc'**

**class** circuits.core.**Bridge** (**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'bridge'**

**init** (*socket*, *channel='bridge'*)

**class** circuits.core.**Debugger** (*errors=True*, *events=True*, *file=None*, *logger=None*, *prefix=None*, *trim=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Create a new Debugger Component

Creates a new Debugger Component that listens to all events in the system printing each event to sys.stderr or a Logger Component.

**Variables**

- ***IgnoreEvents*** – list of events (str) to ignore
- ***IgnoreChannels*** – list of channels (str) to ignore
- **enabled** – Enabled/Disabled flag

**Parameters** **log** – Logger Component instance or None (*default*)

initializes x; see x.__class__.__doc__ for signature

**IgnoreChannels = []**

**IgnoreEvents = ['generate_events']**

**class** circuits.core.**Timer** (*interval*, *event*, **channels*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Timer Component

A timer is a component that fires an event once after a certain delay or periodically at a regular interval.

**Parameters**

- **interval** (`datetime` or number of seconds as a `float`) – the delay or interval to wait for until the event is fired. If interval is specified as datetime, the interval is recalculated as the time span from now to the given datetime.
- **event** (*Event*) – the event to fire.

- **persist** (bool) – An optional keyword argument which if True will cause the event to be fired repeatedly once per configured interval until the timer is unregistered. **Default:** False

**expiry**

**reset**(*interval=None*)
Reset the timer, i.e. clear the amount of time already waited for.

**class** circuits.core.**Manager**(**args*, ***kwargs*)
Bases: object

The manager class has two roles. As a base class for component implementation, it provides methods for event and handler management. The method *fireEvent()* appends a new event at the end of the event queue for later execution. *waitEvent()* suspends the execution of a handler until all handlers for a given event have been invoked. *callEvent()* combines the last two methods in a single method.

The methods *addHandler()* and *removeHandler()* allow handlers for events to be added and removed dynamically. (The more common way to register a handler is to use the *handler()* decorator or derive the class from *Component*.)

In its second role, the *Manager* takes the role of the event executor. Every component hierarchy has a root component that maintains a queue of events. Firing an event effectively means appending it to the event queue maintained by the root manager. The *flush()* method removes all pending events from the queue and, for each event, invokes all the handlers. Usually, *flush()* is indirectly invoked by *run()*.

The manager optionally provides information about the execution of events as automatically generated events. If an *Event* has its success attribute set to True, the manager fires a Success event if all handlers have been executed without error. Note that this event will be enqueued (and dispatched) immediately after the events that have been fired by the event's handlers. So the success event indicates both the successful invocation of all handlers for the event and the processing of the immediate follow-up events fired by those handlers.

Sometimes it is not sufficient to know that an event and its immediate follow-up events have been processed. Rather, it is important to know when all state changes triggered by an event, directly or indirectly, have been performed. This also includes the processing of events that have been fired when invoking the handlers for the follow-up events and the processing of events that have again been fired by those handlers and so on. The completion of the processing of an event and all its direct or indirect follow-up events may be indicated by a Complete event. This event is generated by the manager if *Event* has its complete attribute set to True.

Apart from the event queue, the root manager also maintains a list of tasks, actually Python generators, that are updated when the event queue has been flushed.

initializes x; see x.__class__.__doc__ for signature

**addHandler**(*f*)

**call**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a yield on the top execution level of a handler (e.g. "yield self.callEvent(event)"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**callEvent**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a yield on the top execution level of a handler (e.g. "yield self.callEvent(event)"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**fire**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**fireEvent**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**flush**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**flushEvents**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**getHandlers**(*event*, *channel*, ***kwargs*)

**join**()

**name**
Return the name of this Component/Manager

**pid**
Return the process id of this Component/Manager

**processTask**(*event*, *task*, *parent=None*)

**registerChild**(*component*)

**registerTask**(*g*)

**removeHandler**(*method*, *event=None*)

**run**(*socket=None*)
Run this manager. The method fires the `Started` event and then continuously calls *`tick()`*.

The method returns when the manager's *`stop()`* method is invoked.

If invoked by a programs main thread, a signal handler for the `INT` and `TERM` signals is installed. This handler fires the corresponding `Signal` events and then calls *`stop()`* for the manager.

**running**
Return the running state of this Component/Manager

**start**(*process=False*, *link=None*)
Start a new thread or process that invokes this manager's `run()` method. The invocation of this method returns immediately after the task or process has been started.

**stop**(*code=None*)
Stop this manager. Invoking this method causes an invocation of `run()` to return.

**tick**(*timeout=-1*)
Execute all possible actions once. Process all registered tasks and flush the event queue. If the application is running fire a GenerateEvents to get new events from sources.

This method is usually invoked from *run()*. It may also be used to build an application specific main loop.

**Parameters** **timeout** (*float, measuring seconds*) – the maximum waiting time spent in this method. If negative, the method may block until at least one action has been taken.

**unregisterChild**(*component*)

**unregisterTask**(*g*)

**wait**(*event*, **channels*, ***kwargs*)

**waitEvent**(*event*, **channels*, ***kwargs*)

**exception** circuits.core.**TimeoutError**
Bases: exceptions.Exception

Raised if wait event timeout occurred

### circuits.io package

#### Submodules

#### circuits.io.events module I/O Events

This module implements commonly used I/O events used by other I/O modules.

**class** circuits.io.events.**accessed**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

accessed Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to True, an associated event success (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'accessed'**

**class** `circuits.io.events.`**close**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

close Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'close'**

**class** `circuits.io.events.`**closed**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

closed Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'closed'**

**class** `circuits.io.events.`**created**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

created Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'created'**

**class** `circuits.io.events.`**deleted**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

deleted Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'deleted'**

**class** circuits.io.events.**eof**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

eof Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'eof'**

**class** circuits.io.events.**error**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

error Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to

as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'error'**

**class** `circuits.io.events.`**modified**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

modified Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'modified'**

**class** `circuits.io.events.`**moved**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

moved Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'moved'**

**class** `circuits.io.events.`**open**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

open Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'open'**

**class** `circuits.io.events.`**opened**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

opened Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'opened'**

**class** circuits.io.events.**read**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

read Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'read'**

**class** circuits.io.events.**ready**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

ready Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'ready'**

**class** circuits.io.events.**seek**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

seek Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'seek'**

**class** `circuits.io.events.`**started**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

started Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'started'**

**class** circuits.io.events.**stopped**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

stopped Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'stopped'**

**class** circuits.io.events.**unmounted**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

unmounted Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'unmounted'**

**class** `circuits.io.events.`**write**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

write Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'write'**

**circuits.io.file module** File I/O

This module implements a wrapper for basic File I/O.

**class** `circuits.io.file.`**File**(**args*, ***kwargs*)
Bases: *circuits.core.components.Component*

initializes x; see x.__class__.__doc__ for signature

**channel = 'file'**

**close**()

**closed**

**filename**

**init**(*filename*, *mode='r'*, *bufsize=4096*, *encoding=None*, *channel='file'*)

**mode**

**seek**(*offset*, *whence=0*)

**write**(*data*)

**circuits.io.notify module**

**circuits.io.process module** Process

This module implements a wrapper for basic `subprocess.Popen` functionality.

**class** `circuits.io.process.`**Process**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'process'**

**init**(*args*, *cwd=None*, *shell=False*)

**kill**()

**signal**(*signal*)

**start**()

**status**

**stop**()

**wait**()

**write**(*data*)

**circuits.io.serial module** Serial I/O

This module implements basic Serial (RS232) I/O.

**class** circuits.io.serial.**Serial**(*port*, *baudrate=115200*, *bufsize=4096*, *timeout=0.2*, *channel='serial'*)
Bases: *circuits.core.components.Component*

**channel = 'serial'**

**close**()

**write**(*data*)

**Module contents** I/O Support

This package contains various I/O Components. Provided are a generic File Component, StdIn, StdOut and StdErr components. Instances of StdIn, StdOut and StdErr are also created by importing this package.

### circuits.net package

**Submodules**

**circuits.net.events module** Networking Events

This module implements commonly used Networking events used by socket components.

**class** circuits.net.events.**broadcast**(**args*)
Bases: *circuits.core.events.Event*

broadcast Event

This Event is used by the UDPServer/UDPClient sockets to send a message on the <broadcast> network.

---

**Note:**

- This event is never sent, it is used to send data.
- This event is used for both Client and Server UDP Components.

---

**Parameters** **args** – (data, port)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'broadcast'**

**class** circuits.net.events.**close**(**args*)
Bases: *circuits.core.events.Event*

close Event

This Event is used to notify a client, client connection or server that we want to close.

---

**Note:**

- This event is never sent, it is used to close.

•This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: () Server: (sock)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'close'**

**class** circuits.net.events.**closed**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

closed Event

This Event is sent when a server has closed its listening socket.

---

**Note:** This event is for Server components.

---

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'closed'**

**class** circuits.net.events.**connect**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

connect Event

This Event is sent when a new client connection has arrived on a server. This event is also used for client's to initiate a new connection to a remote host.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters**

- **args** (*tuple*) – Client: (host, port) Server: (sock, host, port)
- **kwargs** (*dict*) – Client: (ssl)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'connect'**

**class** circuits.net.events.**connected**(*host*, *port*)
Bases: *circuits.core.events.Event*

connected Event

This Event is sent when a client has successfully connected.

---

**Note:** This event is for Client Components.

---

**Parameters**

- **host** – The hostname connected to.
- **port** – The port connected to

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'connected'**

**class** circuits.net.events.**disconnect**(**args*)
Bases: *circuits.core.events.Event*

disconnect Event

This Event is sent when a client connection has closed on a server. This event is also used for client's to disconnect from a remote host.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: () Server: (sock)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'disconnect'**

**class** circuits.net.events.**disconnected**
Bases: *circuits.core.events.Event*

disconnected Event

This Event is sent when a client has disconnected

---

**Note:** This event is for Client Components.

---

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'disconnected'**

**class** circuits.net.events.**error**(**args*)
Bases: *circuits.core.events.Event*

error Event

This Event is sent when a client or server connection has an error.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (error) Server: (sock, error)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'error'**

**class** circuits.net.events.**read**(**args*)
Bases: *circuits.core.events.Event*

read Event

This Event is sent when a client or server connection has read any data.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (data) Server: (sock, data)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'read'**

**class** circuits.net.events.**ready**(*component*, *bind=None*)
Bases: *circuits.core.events.Event*

ready Event

This Event is used to notify the rest of the system that the underlying Client or Server Component is ready to begin processing connections or incoming/outgoing data. (This is triggered as a direct result of having the capability to support multiple client/server components with a single poller component instance in a system).

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters**

- **component** – The Client/Server Component that is ready.
- **bind** – The (host, port) the server has bound to.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'ready'**

**class** circuits.net.events.**unreachable**(*host*, *port*, *reason=None*)
Bases: *circuits.core.events.Event*

unreachable Event

This Event is sent when a server is unreachable for a client

**Parameters**

- **host** – Server hostname or IP
- **port** – Server port

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'unreachable'**

**class** circuits.net.events.**write**(**args*)
Bases: *circuits.core.events.Event*

write Event

This Event is used to notify a client, client connection or server that we have data to be written.

---

**Note:**

•This event is never sent, it is used to send data.

•This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (data) Server: (sock, data)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'write'**

**circuits.net.sockets module** Socket Components

This module contains various Socket Components for use with Networking.

**class** circuits.net.sockets.**Client**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'client'**

**close**()

**connected**

**parse_bind_parameter**(*bind_parameter*)

**write**(*data*)

circuits.net.sockets.**Pipe**(**channels*, ***kwargs*)
Create a new full duplex Pipe

Returns a pair of UNIXClient instances connected on either side of the pipe.

**class** circuits.net.sockets.**Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'server'**

**close**(*sock=None*)

**connected**

**host**

**parse_bind_parameter**(*bind_parameter*)

**port**

**write**(*sock*, *data*)

**class** circuits.net.sockets.**TCP6Client**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.TCPClient*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

**class** circuits.net.sockets.**TCP6Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.TCPServer*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

**class** circuits.net.sockets.**TCPClient**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.Client*

**connect**(*host*, *port*, *secure=False*, ***kwargs*)

**init**(*connect_timeout=5*, **args*, ***kwargs*)

**socket_family = 2**

**class** circuits.net.sockets.**TCPServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 2**

circuits.net.sockets.**UDP6Client**
alias of *UDP6Server*

**class** circuits.net.sockets.**UDP6Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.UDPServer*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

circuits.net.sockets.**UDPClient**
alias of *UDPServer*

**class** circuits.net.sockets.**UDPServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

**broadcast**(*data*, *port*)

**close**()

**socket_family = 2**

**write**(*address*, *data*)

**class** circuits.net.sockets.**UNIXClient**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.Client*

**connect**(*path*, *secure=False*, ***kwargs*)

**ready**(*component*)

**class** circuits.net.sockets.**UNIXServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

circuits.net.sockets.**do_handshake**(*sock*, *on_done=None*, *on_error=None*, *extra_args=None*)
SSL Async Handshake

**Parameters**

- **on_done** – Function called when handshake is complete
- **on_error** – Function called when handshake errored

circuits.net.sockets.**parse_ipv4_parameter**(*bind_parameter*)

circuits.net.sockets.**parse_ipv6_parameter**(*bind_parameter*)

**Module contents** Networking Components

This package contains components that implement network sockets and protocols for implementing client and server network applications.

### circuits.node package

#### Submodules

#### circuits.node.client module

**class** circuits.node.client.**Client**(*host*, *port*, *channel='node_client'*, *receive_event_firewall=None*, *send_event_firewall=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Node Client (peer)

Create new connection for a node.

**Parameters**

- **hostname** (*str*) – hostname to connect.
- **port** (*int*) – port to connect.
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** `node_client`
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for sending. **Default:** `None` (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for executing **Default:** `None` (no firewall)

**channel = 'node_client'**

**close**()
Close the connection

**connect**()
Create the connection

**send**(*event*)
Send event through the connection

**Parameters** **event** (*circuits.core.events.Event*) – Event to send.

**Returns** The result object of the sended event

**Return type** generator

**circuits.node.events module**

**class** circuits.node.events.**connected_to**(*connection_name*, *hostname*, *port*, *client_channel*, *client_obj*)

Bases: *circuits.core.events.Event*

Connected to a peer

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote system.
- **port** (*int*) – connection port of the remote system.
- **client_channel** (*str*) – Channel used for client event.
- **client_obj** (*circuits.net.sockets.Client*) – Client object.

**name = 'connected_to'**

**class** circuits.node.events.**disconnected_from**(*connection_name*, *hostname*, *port*, *client_channel*, *client_obj*)

Bases: *circuits.node.events.connected_to*

Disconnected from a peer

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote system.
- **port** (*int*) – connection port of the remote system.
- **client_channel** (*str*) – Channel used for client event.
- **client_obj** (*circuits.net.sockets.Client*) – Client object.

**name = 'disconnected_from'**

**class** circuits.node.events.**remote**(*event*, *connection_name*, *channel=None*)

Bases: *circuits.core.events.Event*

send event to a peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **connection_name** (*str*) – Connection name.
- **channel** (*str*) – Remote channel (channel to use on peer).

**name = 'remote'**

**circuits.node.node module** Node

this module manage node (start server, add peer, ...) .. seealso:: Examples in **:dir:'examples.node'**

**class** circuits.node.node.**Node**(*port=None*, *channel='node'*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

this class manage node (start server, add peer, ...) .. seealso:: Examples in **:dir:'examples.node'**

Start node system.

**Parameters**

- **port** (*int*) – An optional keyword argument which if defined, start server on this port. **Default:** None (don't start the server)
- **server_ip** (*str*) – An optional keyword argument which define ip where the socket has listen to. **Default:** 0.0.0.0 (all ip is allowed)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** node
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for sending. **Default:** None (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for executing **Default:** None (no firewall)

**add**(*connection_name*, *hostname*, *port*, ***kwargs*)
Add new peer to the node.

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote node.
- **port** (*int*) – port of the remote node.
- **auto_remote_event** (*dict*) – An optional keyword argument which if defined, bind events automatically to remote execution. **Default:** {} (no events)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for client event. If this keyword is not defined the method will generate the channel name automatically.
- **reconnect_delay** (*int*) – An optional keyword argument which if defined, set auto reconnect delay. **Default:** 10 (seconde)
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for sending. **Default:** None (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, setfunction or method to call to check if event is allowed for executing **Default:** None (no firewall)

**Returns** Channel used on client event.

**Return type** str

**channel = 'node'**

**get_connection_names**()
Get connections names

**Returns** The list of connections names

**Return type** list of str

**get_peer**(*connection_name*)
Get a client object by name

**Parameters** **connection_name** (*str*) – Connection name.

**Returns** The Client object

**Return type** *circuits.node.client.Client*

**circuits.node.server module**

**class** circuits.node.server.**Server**(*port*, *server_ip='0.0.0.0'*, *channel='node'*, *receive_event_firewall=None*, *send_event_firewall=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Node server.

Create server on node system.

**Parameters**

- **port** (*int*) – start server on this port.
- **server_ip** (*str*) – An optional keyword argument which which define ip where the socket has listen to. **Default:** `0.0.0.0` (all ip is allowed)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** `node`
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for sending **Default:** `None` (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for executing **Default:** `None` (no firewall)

**channel = 'node'**

**get_socks**()
Get clients sockets list

**Returns** The list of client socket

**Return type** list of `socket.socket`

**host**

**port**

**send**(*event*, *sock*, *no_result=False*)
Send event to peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **sock** (`socket.socket`) – Client's socket (peer selection).
- **no_result** (*bool*) – An optional keyword argument which if True don't return the event result. **Default:** `False` (wait the result)

**Returns** The result of remote event

**Return type** generator

**send_all**(*event*)
Send event to all peer

**Parameters** **event** (*circuits.core.events.Event*) – Event to execute remotely.

**send_to**(*event*, *socks*)
Send event to multiple peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **socks** (list of socket.socket) – Client's socket list (peer selection).

### circuits.node.utils module

circuits.node.utils.**dump_event**(*e*, *id*)
circuits.node.utils.**dump_value**(*v*)

circuits.node.utils.**load_event**(*s*)

circuits.node.utils.**load_value**(*v*)

**Module contents** Node

Distributed processing support for circuits.

## circuits.protocols package

### Submodules

### circuits.protocols.http module

**class** circuits.protocols.http.**HTTP**(*encoding='utf-8'*, *channel='web'*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.protocols.http.**ResponseObject**(*headers*, *status*, *version*)
Bases: object

**read**()

**class** circuits.protocols.http.**request**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

request Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'request'**

**class** `circuits.protocols.http.`**response**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

response Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'response'**

**circuits.protocols.irc module** Internet Relay Chat Protocol

This package implements the Internet Relay Chat Protocol or commonly known as IRC. Support for both server and client is implemented.

**circuits.protocols.line module** Line Protocol

This module implements the basic Line protocol.

This module can be used in both server and client implementations.

**class** `circuits.protocols.line.`**Line**(**args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

Line Protocol

Implements the Line Protocol.

Incoming data is split into lines with a splitter function. For each line of data processed a Line Event is created. Any unfinished lines are appended into an internal buffer.

A custom line splitter function can be passed to customize how data is split into lines. This function must accept two arguments, the data to process and any left over data from a previous invocation of the splitter function. The function must also return a tuple of two items, a list of lines and any left over data.

**Parameters** **splitter** (*function*) – a line splitter function

This Component operates in two modes. In normal operation it's expected to be used in conjunction with components that expose a Read Event on a "read" channel with only one argument (data). Some builtin components that expose such events are: - circuits.net.sockets.TCPClient - circuits.io.File

The second mode of operation works with circuits.net.sockets.Server components such as TCPServer, UNIXServer, etc. It's expected that two arguments exist in the Read Event, sock and data. The following two arguments can be passed to affect how unfinished data is stored and retrieved for such components:

**Parameters** **getBuffer** (*function*) – function to retrieve the buffer for a client sock

This function must accept one argument (sock,) the client socket whoose buffer is to be retrieved.

**Parameters** **updateBuffer** (*function*) – function to update the buffer for a client sock

This function must accept two arguments (sock, buffer,) the client socket and the left over buffer to be updated.

**@note: This Component must be used in conjunction with a Component that** exposes Read events on a "read" Channel.

initializes x; see x.__class__.__doc__ for signature

**class** circuits.protocols.line.**line**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

line Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'line'**

circuits.protocols.line.**splitLines**(*s*, *buffer*) → lines, buffer

Append s to buffer and find any new lines of text in the string splitting at the standard IRC delimiter CRLF. Any new lines found, return them as a list and the remaining buffer for further processing.

**circuits.protocols.websocket module**

**class** circuits.protocols.websocket.**WebSocketCodec**(*sock=None*, *data=bytearray(b'')*, **args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

WebSocket Protocol

Implements the Data Framing protocol for WebSocket.

This component is used in conjunction with a parent component that receives Read events on its channel. When created (after a successful WebSocket setup handshake), the codec registers a handler on the parent's channel that filters out these Read events for a given socket (if used in a server) or all Read events (if used in a client). The data is decoded and the contained payload is emitted as Read events on the codec's channel.

The data from write events sent to the codec's channel (with socket argument if used in a server) is encoded according to the WebSocket Data Framing protocol. The encoded data is then forwarded as write events on the parents channel.

Creates a new codec.

**Parameters** **sock** – the socket used in Read and write events (if used in a server, else None)

**channel = 'ws'**

**Module contents** Networking Protocols

This package contains components that implement various networking protocols.

### circuits.tools package

**Module contents** Circuits Tools

circuits.tools contains a standard set of tools for circuits. These tools are installed as executables with a prefix of "circuits."

circuits.tools.**tryimport**(*modules*, *obj=None*, *message=None*)

circuits.tools.**walk**(*x*, *f*, *d=0*, *v=None*)

circuits.tools.**edges**(*x*, *e=None*, *v=None*, *d=0*)

circuits.tools.**findroot**(*x*)

circuits.tools.**kill**(*x*)

circuits.tools.**graph**(*x*, *name=None*)
Display a directed graph of the Component structure of x

**Parameters**

- **x** (*Component or Manager*) – A Component or Manager to graph
- **name** (*str*) – A name for the graph (defaults to x's name)

@return: A directed graph representing x's Component structure. @rtype: str

circuits.tools.**inspect**(*x*)
Display an inspection report of the Component or Manager x

**Parameters** **x** (*Component or Manager*) – A Component or Manager to graph

@return: A detailed inspection report of x @rtype: str

circuits.tools.**deprecated**(*f*)

### circuits.web package

**Subpackages**

**circuits.web.dispatchers package**

**Submodules**

**circuits.web.dispatchers.dispatcher module** Dispatcher

This module implements a basic URL to Channel dispatcher. This is the default dispatcher used by circuits.web

**class** circuits.web.dispatchers.dispatcher.**Dispatcher**(***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

circuits.web.dispatchers.dispatcher.**find_handlers**(*req*, *paths*)

circuits.web.dispatchers.dispatcher.**resolve_methods**(*parts*)

circuits.web.dispatchers.dispatcher.**resolve_path**(*paths*, *parts*)

**circuits.web.dispatchers.jsonrpc module** JSON RPC

This module implements a JSON RPC dispatcher that translates incoming RPC calls over JSON into RPC events.

**class** circuits.web.dispatchers.jsonrpc.**JSONRPC**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.web.dispatchers.jsonrpc.**rpc**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

RPC Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'rpc'**

**circuits.web.dispatchers.static module** Static

This modStatic implements a Static dispatcher used to serve up static resources and an optional apache-style directory listing.

**class** circuits.web.dispatchers.static.**Static**(*path=None*, *docroot=None*, *defaults=('index.html', 'index.xhtml')*, *dirlisting=False*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**circuits.web.dispatchers.virtualhosts module** VirtualHost

This module implements a virtual host dispatcher that sends requests for configured virtual hosts to different dispatchers.

**class** circuits.web.dispatchers.virtualhosts.**VirtualHosts**(*domains*)

Bases: *circuits.core.components.BaseComponent*

Forward to anotehr Dispatcher based on the Host header.

This can be useful when running multiple sites within one server. It allows several domains to point to different parts of a single website structure. For example: - http://www.domain.example -> / - http://www.domain2.example -> /domain2 - http://www.domain2.example:443 -> /secure

**Parameters** **domains** (*dict*) – a dict of {host header value: virtual prefix} pairs.

The incoming "Host" request header is looked up in this dict, and, if a match is found, the corresponding "virtual prefix" value will be prepended to the URL path before passing the request onto the next dispatcher.

Note that you often need separate entries for "example.com" and "www.example.com". In addition, "Host" headers may contain the port number.

**channel = 'web'**

**circuits.web.dispatchers.xmlrpc module** XML RPC

This module implements a XML RPC dispatcher that translates incoming RPC calls over XML into RPC events.

**class** circuits.web.dispatchers.xmlrpc.**XMLRPC**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.web.dispatchers.xmlrpc.**rpc**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

rpc Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'rpc'**

**Module contents** Dispatchers

This package contains various circuits.web dispatchers By default a `circuits.web.Server` Component uses the `dispatcher.Dispatcher`

### circuits.web.parsers package

#### Submodules

#### circuits.web.parsers.http module

**class** `circuits.web.parsers.http.`**HttpParser**(*kind=2*, *decompress=False*)

Bases: `object`

**execute**(*data*, *length*)

**get_headers**()

**get_method**()

**get_path**()

**get_query_string**()

**get_scheme**()

**get_status_code**()

**get_url**()

**get_version**()

**is_chunked**()
return True if Transfer-Encoding header value is chunked

**is_headers_complete**()
return True if all headers have been parsed.

**is_message_begin**()
return True if the parsing start

**is_message_complete**()
return True if the parsing is done (we get EOF)

**is_partial_body**()
return True if a chunk of body have been parsed

**is_upgrade**()
Do we get upgrade header in the request. Useful for websockets

**recv_body**()
return last chunk of the parsed body

**recv_body_into**(*barray*)
Receive the last chunk of the parsed bodyand store the data in a buffer rather than creating a new string.

**should_keep_alive**()
return True if the connection should be kept alive

**exception** circuits.web.parsers.http.**InvalidChunkSize**
Bases: exceptions.Exception

error raised when we parse an invalid chunk size

**exception** circuits.web.parsers.http.**InvalidHeader**
Bases: exceptions.Exception

error raised on invalid header

**exception** circuits.web.parsers.http.**InvalidRequestLine**
Bases: exceptions.Exception

error raised when first line is invalid

### circuits.web.parsers.multipart module

**Parser for multipart/form-data** This module provides a parser for the multipart/form-data format. It can read from a file, a socket or a WSGI environment. The parser can be used to replace cgi.FieldStorage (without the bugs) and works with Python 2.5+ and 3.x (2to3).

#### Licence (MIT)

Copyright (c) 2010, Marcel Hellkamp. Inspired by the Werkzeug library: http://werkzeug.pocoo.org/

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**class** circuits.web.parsers.multipart.**MultiDict**(**a*, ***k*)
Bases: _abcoll.MutableMapping

A dict that remembers old values for each key

**append**(*key*, *value*)

**get**(*key*, *default=None*, *index=-1*)

**getall**(*key*)

**iterallitems**()

**keys**()

**replace**(*key*, *value*)

**exception** circuits.web.parsers.multipart.**MultipartError**
Bases: exceptions.ValueError

**class** circuits.web.parsers.multipart.**MultipartParser**(*stream*, *boundary*, *content_length=-1*, *disk_limit=1073741824*, *mem_limit=1048576*, *memfile_limit=262144*, *buffer_size=65536*, *charset='latin1'*)
Bases: object

Parse a multipart/form-data byte stream. This object is an iterator over the parts of the message.

**Parameters**

- **stream** – A file-like stream. Must implement .read(size).
- **boundary** – The multipart boundary as a byte string.
- **content_length** – The maximum number of bytes to read.

**get**(*name*, *default=None*)
Return the first part with that name or a default value (None).

**get_all**(*name*)
Return a list of parts with that name.

**parts**()
Returns a list with all parts of the multipart message.

**class** circuits.web.parsers.multipart.**MultipartPart**(*buffer_size=65536*, *memfile_limit=262144*, *charset='latin1'*)
Bases: object

**feed**(*line*, *nl=''*)

**finish_header**()

**is_buffered**()
Return true if the data is fully buffered in memory.

**save_as**(*path*)

**value**
Data decoded with the specified charset

**write_body**(*line*, *nl*)

**write_header**(*line*, *nl*)

circuits.web.parsers.multipart.**copy_file**(*stream*, *target*, *maxread=-1*, *buffer_size=32*)
Read from :stream and write to :target until :maxread or EOF.

circuits.web.parsers.multipart.**header_quote**(*val*)

circuits.web.parsers.multipart.**header_unquote**(*val*, *filename=False*)

circuits.web.parsers.multipart.**parse_form_data**(*environ*, *charset='utf8'*, *strict=False*, ***kw*)
Parse form data from an environ dict and return a (forms, files) tuple. Both tuple values are dictionaries with the form-field name as a key (text_type) and lists as values (multiple values per key are possible). The forms-dictionary contains form-field values as text_type strings. The files-dictionary contains *MultipartPart* instances, either because the form-field was a file-upload or the value is to big to fit into memory limits.

**Parameters**

- **environ** – An WSGI environment dict.
- **charset** – The charset to use if unsure. (default: utf8)
- **strict** – If True, raise *MultipartError* on any parsing errors. These are silently ignored by default.

circuits.web.parsers.multipart.**parse_options_header**(*header*, *options=None*)

circuits.web.parsers.multipart.**tob**(*data*, *enc='utf8'*)

### circuits.web.parsers.querystring module

**class** circuits.web.parsers.querystring.**QueryStringParser**(*data*)
Bases: object

**parse**(*key*, *value*)

**process**(*pair*)

**tokens**(*key*)

**class** circuits.web.parsers.querystring.**QueryStringToken**
Bases: object

**ARRAY = 'ARRAY'**

**KEY = 'KEY'**

**OBJECT = 'OBJECT'**

**Module contents** circuits.web parsers

### circuits.web.websockets package

### Submodules

#### circuits.web.websockets.client module

**class** circuits.web.websockets.client.**WebSocketClient**(*url*, *channel='wsclient'*, *wschannel='ws'*, *headers=None*)

Bases: *circuits.core.components.BaseComponent*

An RFC 6455 compliant WebSocket client component. Upon receiving a circuits.web.client.Connect event, the component tries to establish the connection to the server in a two stage process. First, a *circuits.net.events.connect* event is sent to a child *TCPClient*. When the TCP connection has been established, the HTTP request for opening the WebSocket is sent to the server. A failure in this setup process is signaled by raising an *NotConnected* exception.

When the server accepts the request, the WebSocket connection is established and can be used very much like an ordinary socket by handling *read* events on and sending *write* events to the channel specified as the wschannel parameter of the constructor. Firing a *close* event on that channel closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

**Parameters**

- **url** – the URL to connect to.
- **channel** – the channel used by this component
- **wschannel** – the channel used for the actual WebSocket communication (read, write, close events)
- **headers** – additional headers to be passed with the WebSocket setup HTTP request

**channel = 'wsclient'**

**close**()

**connected**

#### circuits.web.websockets.dispatcher module

**class** circuits.web.websockets.dispatcher.**WebSocketsDispatcher**(*path=None*, *wschannel='wsserver'*, **args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

This class implements an RFC 6455 compliant WebSockets dispatcher that handles the WebSockets handshake and upgrades the connection.

The dispatcher listens on its channel for Request events and tries to match them with a given path. Upon a match, the request is checked for the proper Opening Handshake information. If successful, the dispatcher confirms the establishment of the connection to the client. Any subsequent data from the client is handled as a WebSocket data frame, decoded and fired as a Read event on the wschannel passed to the constructor. The data from *write* events on that channel is encoded as data frames and forwarded to the client.

Firing a Close event on the wschannel closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

**Parameters**

- **path** – the path to handle. Requests that start with this path are considered to be WebSocket Opening Handshakes.
- **wschannel** – the channel on which read events from the client will be delivered and where *write* events to the client will be sent to.

**channel = 'web'**

**Module contents** circuits.web websockets

**Submodules**

**circuits.web.client module**

**class** circuits.web.client.**Client**(*channel='client'*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'client'**

**close**()

**connect**(*event*, *host=None*, *port=None*, *secure=None*)

**connected**

**request**(*method*, *url*, *body=None*, *headers=None*)

**response**

**write**(*data*)

**exception** circuits.web.client.**HTTPException**

Bases: exceptions.Exception

**exception** circuits.web.client.**NotConnected**

Bases: *circuits.web.client.HTTPException*

circuits.web.client.**parse_url**(*url*)

**class** circuits.web.client.**request**(*method*, *path*, *body=None*, *headers=None*)

Bases: *circuits.core.events.Event*

request Event

This Event is used to initiate a new request.

**Parameters**

- **method** (*str*) – HTTP Method (PUT, GET, POST, DELETE)
- **url** (*str*) – Request URL

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'request'**

**circuits.web.constants module** Global Constants

This module implements required shared global constants.

**circuits.web.controllers module** Controllers

This module implements ...

**class** circuits.web.controllers.**BaseController**(**args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = '/'**

**expires**(*secs=0*, *force=False*)

**forbidden** (*description=None*)
Return a 403 (Forbidden) response

**Parameters** **description** (*str*) – Message to display

**notfound** (*description=None*)
Return a 404 (Not Found) response

**Parameters** **description** (*str*) – Message to display

**redirect** (*urls*, *code=None*)
Return a 30x (Redirect) response

Redirect to another location specified by urls with an optional custom response code.

**Parameters**

- **urls** (*str or list*) – A single URL or list of URLs
- **code** (*int*) – HTTP Redirect code

**serve_download** (*path*, *name=None*)

**serve_file** (*path*, *type=None*, *disposition=None*, *name=None*)

**uri**
Return the current Request URI

**See also:**

*circuits.web.url.URL*

**class** circuits.web.controllers.**Controller** (**args*, ***kwargs*)
Bases: *circuits.web.controllers.BaseController*

initializes x; see x.__class__.__doc__ for signature

**class** circuits.web.controllers.**ExposeJSONMetaClass** (*name*, *bases*, *dct*)
Bases: type

**class** circuits.web.controllers.**ExposeMetaClass** (*name*, *bases*, *dct*)
Bases: type

**class** circuits.web.controllers.**JSONController** (**args*, ***kwargs*)
Bases: *circuits.web.controllers.BaseController*

initializes x; see x.__class__.__doc__ for signature

circuits.web.controllers.**expose** (**channels*, ***config*)

circuits.web.controllers.**exposeJSON** (**channels*, ***config*)

**circuits.web.errors module** Errors

This module implements a set of standard HTTP Errors.

**class** circuits.web.errors.**forbidden** (*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Forbidden error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 403**

**name = 'forbidden'**

**class** circuits.web.errors.**httperror**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.core.events.Event*

An event for signaling an HTTP error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 500**

**description = ''**

**name = 'httperror'**

**sanitize**()

**class** circuits.web.errors.**notfound**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Not Fouond error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 404**

**name = 'notfound'**

**class** circuits.web.errors.**redirect**(*request*, *response*, *urls*, *code=None*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Redirect response

The constructor creates a new instance and modifies the *response* argument to reflect a redirect response to the given *url*.

**name = 'redirect'**

**class** circuits.web.errors.**unauthorized**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Unauthorized error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 401**

**name = 'unauthorized'**

**circuits.web.events module** Events

This module implements the necessary Events needed.

**class** circuits.web.events.**request**(*Event*) → request Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'request'**

**success = True**

**class** `circuits.web.events.`**response**(*Event*) → response Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'response'**

**success = True**

**class** `circuits.web.events.`**stream**(*Event*) → stream Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'stream'**

**success = True**

**class** circuits.web.events.**terminate**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

terminate Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'terminate'**

**circuits.web.exceptions module** Exceptions

This module implements a set of standard HTTP Errors as Python Exceptions.

Note: This code is mostly borrowed from werkzeug and adapted for circuits.web

**exception** circuits.web.exceptions.**Redirect**(*urls*, *status=None*)
Bases: *circuits.web.exceptions.HTTPException*

**code = 303**

**exception** circuits.web.exceptions.**RequestTimeout**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*408 Request Timeout*

Raise to signalize a timeout.

**code = 408**

**description = " The server closed the network connection because the browser didn't finish the request within the**

**exception** circuits.web.exceptions.**RequestEntityTooLarge**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*413 Request Entity Too Large*

The status code one should return if the data submitted exceeded a given limit.

**code = 413**

**description = ' The data value transmitted exceeds the capacity limit. '**

**exception** circuits.web.exceptions.**ServiceUnavailable**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*503 Service Unavailable*

Status code you should return if a service is temporarily unavailable.

**code = 503**

**description = ' The server is temporarily unable to service your request due to maintenance downtime or capacity**

**exception** circuits.web.exceptions.**UnsupportedMediaType**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*415 Unsupported Media Type*

The status code returned if the server is unable to handle the media type the client transmitted.

**code = 415**

**description = ' The server does not support the media type transmitted in the request. '**

**exception** circuits.web.exceptions.**RequestURITooLarge**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*414 Request URI Too Large*

Like *413* but for too long URLs.

**code = 414**

**description = ' The length of the requested URL exceeds the capacity limit for this server. The request cannot be p**

**exception** circuits.web.exceptions.**BadGateway**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*502 Bad Gateway*

If you do proxying in your application you should return this status code if you received an invalid response from the upstream server it accessed in attempting to fulfill the request.

**code = 502**

**description = ' The proxy server received an invalid response from an upstream server. '**

**exception** circuits.web.exceptions.**HTTPException**(*description=None*, *traceback=None*)
Bases: exceptions.Exception

Baseclass for all HTTP exceptions. This exception can be called by WSGI applications to render a default error page or you can catch the subclasses of it independently and render nicer error messages.

**code = None**

**description = None**

**name**
The status name.

**traceback = True**

**exception** circuits.web.exceptions.**UnicodeError**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

raised by the request functions if they were unable to decode the incoming data properly.

**exception** circuits.web.exceptions.**NotAcceptable**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*406 Not Acceptable*

Raise if the server can't return any content conforming to the *Accept* headers of the client.

**code = 406**

**description = ' The resource identified by the request is only capable of generating response entities which have co**

**exception** circuits.web.exceptions.**BadRequest**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*400 Bad Request*

Raise if the browser sends something to the application the application or server cannot handle.

**code = 400**

**description = ' The browser (or proxy) sent a request that this server could not understand. '**

**exception** circuits.web.exceptions.**RangeUnsatisfiable**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*416 Range Unsatisfiable*

The status code returned if the server is unable to satisfy the request range

**code = 416**

**description = ' The server cannot satisfy the request range(s). '**

**exception** circuits.web.exceptions.**Unauthorized**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*401 Unauthorized*

Raise if the user is not authorized. Also used if you want to use HTTP basic auth.

**code = 401**

**description = " The server could not verify that you are authorized to access the URL requested. You either suppli**

**exception** circuits.web.exceptions.**Gone**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*410 Gone*

Raise if a resource existed previously and went away without new location.

**code = 410**

**description = ' The requested URL is no longer available on this server and there is no forwarding address. <p**

**exception** circuits.web.exceptions.**PreconditionFailed**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*412 Precondition Failed*

Status code used in combination with `If-Match`, `If-None-Match`, or `If-Unmodified-Since`.

**code = 412**

**description = ' The precondition on the request for the URL failed positive evaluation. '**

**exception** circuits.web.exceptions.**InternalServerError**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*500 Internal Server Error*

Raise if an internal server error occurred. This is a good fallback if an unknown error occurred in the dispatcher.

**code = 500**

**description = ' The server encountered an internal error and was unable to complete your request. Either the serv**

**exception** circuits.web.exceptions.**MethodNotAllowed**(*method*, *description=None*)
Bases: *circuits.web.exceptions.HTTPException*

*405 Method Not Allowed*

Raise if the server used a method the resource does not handle. For example *POST* if the resource is view only. Especially useful for REST.

The first argument for this exception should be a list of allowed methods. Strictly speaking the response would be invalid if you don't provide valid methods in the header which you can do with that list.

**code = 405**

**exception** circuits.web.exceptions.**NotFound**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*404 Not Found*

Raise if a resource does not exist and never existed.

**code = 404**

**description = ' The requested URL was not found on the server. If you entered the URL manually please c**

**exception** circuits.web.exceptions.**LengthRequired**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*411 Length Required*

Raise if the browser submitted data but no `Content-Length` header which is required for the kind of processing the server does.

**code = 411**

**description = ' A request with this method requires a valid <code>Content-Length</code> header. '**

**exception** `circuits.web.exceptions.`**`Forbidden`**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*403 Forbidden*

Raise if the user doesn't have the permission for the requested resource but was authenticated.

**code = 403**

**description = " You don't have the permission to access the requested resource. It is either read-protected or not r**

**exception** `circuits.web.exceptions.`**`NotImplemented`**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*501 Not Implemented*

Raise if the application does not support the action requested by the browser.

**code = 501**

**description = ' The server does not support the action requested by the browser. '**

**circuits.web.headers module** Headers Support

This module implements support for parsing and handling headers.

**class** `circuits.web.headers.`**`AcceptElement`**(*value*, *params=None*)
Bases: *circuits.web.headers.HeaderElement*

An element (with parameters) from an Accept* header's element list.

AcceptElement objects are comparable; the more-preferred object will be "less than" the less-preferred object. They are also therefore sortable; if you sort a list of AcceptElement objects, they will be listed in priority order; the most preferred value will be first. Yes, it should have been the other way around, but it's too late to fix now.

**classmethod `from_str`**(*elementstr*)

**`qvalue`**
The qvalue, or priority, of this value.

**class** `circuits.web.headers.`**`CaseInsensitiveDict`**(**args*, ***kwargs*)
Bases: `dict`

A case-insensitive dict subclass.

Each key is changed on entry to str(key).title().

**classmethod `fromkeys`**(*seq*, *value=None*)

**`get`**(*key*, *default=None*)

**`pop`**(*key*, *default=None*)

**`setdefault`**(*key*, *x=None*)

**`update`**(*E*)

**class** circuits.web.headers.**HeaderElement**(*value*, *params=None*)
Bases: object

An element (with parameters) from an HTTP header's element list.

**classmethod from_str**(*elementstr*)
Construct an instance from a string of the form 'token;key=val'.

**static parse**(*elementstr*)
Transform 'token;key=val' to ('token', {'key': 'val'}).

**class** circuits.web.headers.**Headers**(**args*, ***kwargs*)
Bases: *circuits.web.headers.CaseInsensitiveDict*

This class implements a storage for headers as key value pairs. The underlying model of a case insensitive dict matches the requirements for headers quite well, because usually header keys are unique. If several values may be associated with a header key, most HTTP headers represent the values as an enumeration using a comma as item separator.

There is, however one exception (currently) to this rule. In order to set several cookies, there should be multiple headers with the same key, each setting one cookie ("Set-Cookie: some_cookie").

This is modeled by having either a string (common case) or a list (cookie case) as value in the underlying dict. In order to allow easy iteration over all headers as they appear in the HTTP request, the items() method expands associated lists of values. So if you have { "Set-Cookie": [ "cookie1", "cookie2" ] }, the items() method returns the two pairs ("Set-Cookie", "cookie1") and ("Set-Cookie", "cookie2"). This is convenient for most use cases. The only drawback is that len(keys()) is not equal to len(items()) for this specialized dict.

**add_header**(*_name*, *_value*, ***_params*)
Extended header setting.

_name is the header field to add. keyword arguments can be used to set additional parameters for the header field, with underscores converted to dashes. Normally the parameter will be added as key="value" unless value is None, in which case only the key will be added.

Example:

h.add_header('content-disposition', 'attachment', filename='bud.gif')

Note that unlike the corresponding 'email.Message' method, this does *not* handle '(charset, language, value)' tuples: all values must be strings or None.

**append**(*key*, *value*)
If a header with the given name already exists, the value is normally appended to the existing value separated by a comma.

If, however, the already existing entry associated key with a value of type list (as is the case for "Set-Cookie"), the new value is appended to that list.

**elements**(*key*)
Return a sorted list of HeaderElements for the given header.

**get_all**(*name*)
Return a list of all the values for the named field.

**items**()

circuits.web.headers.**header_elements**(*fieldname*, *fieldvalue*)
Return a sorted HeaderElement list.

Returns a sorted HeaderElement list from a comma-separated header string.

**circuits.web.http module** Hyper Text Transfer Protocol

This module implements the server side Hyper Text Transfer Protocol or commonly known as HTTP.

**class** circuits.web.http.**HTTP**(*server*, *encoding='utf-8'*, *channel='web'*)
Bases: *circuits.core.components.BaseComponent*

HTTP Protocol Component

Implements the HTTP server protocol and parses and processes incoming HTTP messages, creating and sending an appropriate response.

The component handles `Read` events on its channel and collects the associated data until a complete HTTP request has been received. It parses the request's content and puts it in a *Request* object and creates a corresponding *Response* object. Then it emits a `Request` event with these objects as arguments.

The component defines several handlers that send a response back to the client.

**base**

**channel = 'web'**

**protocol**

**scheme**

**version**

**circuits.web.loggers module** Logger Component

This module implements Logger Components.

**class** circuits.web.loggers.**Logger**(*file=None*, *logger=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**format = '%(h)s %(l)s %(u)s %(t)s "%(r)s" %(s)s %(b)s "%(f)s" "%(a)s"'**

**log**(*response*)

**log_response**(*response_event*, *value*)

circuits.web.loggers.**formattime**()

**circuits.web.main module** Main

circutis.web Web Server and Testing Tool.

**class** circuits.web.main.**Authentication**(*channel='web'*, *realm=None*, *passwd=None*)
Bases: *circuits.core.components.Component*

**channel = 'web'**

**realm = 'Secure Area'**

**request**(*event*, *request*, *response*)

**users = {'admin': '21232f297a57a5a743894a0e4a801fc3'}**

**class** circuits.web.main.**HelloWorld**(**args*, ***kwargs*)
Bases: *circuits.core.components.Component*

initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**request**(*request*, *response*)

**class** circuits.web.main.**Root**(**args*, ***kwargs*)
Bases: *circuits.web.controllers.Controller*

initializes x; see x.__class__.__doc__ for signature

**hello**(*event*, **args*, ***kwargs*)

circuits.web.main.**main**()

circuits.web.main.**parse_bind**(*bind*)

circuits.web.main.**parse_options**()

circuits.web.main.**select_poller**(*poller*)

### circuits.web.processors module

circuits.web.processors.**process**(*request*, *params*)
circuits.web.processors.**process_multipart**(*request*, *params*)

circuits.web.processors.**process_urlencoded**(*request*, *params*, *encoding='utf-8'*)

### circuits.web.servers module

Web Servers

This module implements the several Web Server components.

**class** circuits.web.servers.**BaseServer**(*bind*, *encoding='utf-8'*, *secure=False*, *certfile=None*, *channel='web'*, *display_banner=True*)
Bases: *circuits.core.components.BaseComponent*

Create a Base Web Server

Create a Base Web Server (HTTP) bound to the IP Address / Port or UNIX Socket specified by the 'bind' parameter.

**Variables** ***server*** – Reference to underlying Server Component

**Parameters** **bind** (*Instance of int, list, tuple or str*) – IP Address / Port or UNIX Socket to bind to.

The 'bind' parameter is quite flexible with what valid values it accepts.

If an int is passed, a TCPServer will be created. The Server will be bound to the Port given by the 'bind' argument and the bound interface will default (normally to "0.0.0.0").

If a list or tuple is passed, a TCPServer will be created. The Server will be bound to the Port given by the 2nd item in the 'bind' argument and the bound interface will be the 1st item.

If a str is passed and it contains the ':' character, this is assumed to be a request to bind to an IP Address / Port. A TCpServer will thus be created and the IP Address and Port will be determined by splitting the string given by the 'bind' argument.

Otherwise if a str is passed and it does not contain the ':' character, a file path is assumed and a UNIXServer is created and bound to the file given by the 'bind' argument.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**display_banner**

**host**

**port**

**secure**

**class** circuits.web.servers.**FakeSock**

**getpeername**()

**class** circuits.web.servers.**Server**(*bind*, ***kwargs*)
Bases: *circuits.web.servers.BaseServer*

Create a Web Server

Create a Web Server (HTTP) complete with the default Dispatcher to parse requests and posted form data dispatching to appropriate Controller(s).

See: circuits.web.servers.BaseServer

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**class** circuits.web.servers.**StdinServer**(*encoding='utf-8'*, *channel='web'*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**host**

**port**

**read**(*data*)

**secure**

**write**(*sock*, *data*)

**circuits.web.sessions module** Session Components

This module implements Session Components that can be used to store and access persistent information.

**class** circuits.web.sessions.**Sessions**(*name='circuits.session'*, *channel='web'*)
Bases: *circuits.core.components.Component*

**channel = 'web'**

**load**(*sid*)

**request**(*request*, *response*)

**save**(*sid*, *data*)
Save User Session Data for sid

circuits.web.sessions.**create_session**(*request*)
Create a unique session id from the request

Returns a unique session using `uuid4()` and a `sha1()` hash of the users IP Address and User Agent in the form of `sid/who`.

circuits.web.sessions.**verify_session**(*request*, *sid*)
Verify a User's Session

This verifies the User's Session by verifying the SHA1 Hash of the User's IP Address and User-Agent match the provided Session ID.

circuits.web.sessions.**who**(*request*, *encoding='utf-8'*)
Create a SHA1 Hash of the User's IP Address and User-Agent

**circuits.web.tools module** Tools

This module implements tools used throughout circuits.web. These tools can also be used within Controlelrs and request handlers.

**class** circuits.web.tools.**ReverseProxy**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**headers = ('X-Real-IP', 'X-Forwarded-For')**

**init**(*headers=None*)
Web Component for identifying the original client IP when a reverse proxy is used

**Parameters** **headers** – List of HTTP headers to read the original client IP

circuits.web.tools.**basic_auth**(*request*, *response*, *realm*, *users*, *encrypt=None*)
Perform Basic Authentication

If auth fails, returns an Unauthorized error with a basic authentication header.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.
- **encrypt** (*callable*) – Callable used to encrypt the password returned from the user-agent. if None it defaults to a md5 encryption.

circuits.web.tools.**check_auth**(*request*, *response*, *realm*, *users*, *encrypt=None*)
Check Authentication

If an Authorization header contains credentials, return True, else False.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.
- **encrypt** (*callable*) – Callable used to encrypt the password returned from the user-agent. if None it defaults to a md5 encryption.

circuits.web.tools.**digest_auth**(*request*, *response*, *realm*, *users*)
Perform Digest Authentication

If auth fails, raise 401 with a digest authentication header.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.

circuits.web.tools.**expires**(*request*, *response*, *secs=0*, *force=False*)
Tool for influencing cache mechanisms using the 'Expires' header.

'secs' must be either an int or a datetime.timedelta, and indicates the number of seconds between response.time and when the response should expire. The 'Expires' header will be set to (response.time + secs).

If 'secs' is zero, the 'Expires' header is set one year in the past, and the following "cache prevention" headers are also set: - 'Pragma': 'no-cache' - 'Cache-Control': 'no-cache, must-revalidate'

If 'force' is False (the default), the following headers are checked: 'Etag', 'Last-Modified', 'Age', 'Expires'. If any are already present, none of the above response headers are set.

circuits.web.tools.**gzip**(*response*, *level=4*, *mime_types=('text/html', 'text/plain')*)
Try to gzip the response body if Content-Type in mime_types.

response.headers['Content-Type'] must be set to one of the values in the mime_types arg before calling this function.

**No compression is performed if any of the following hold:**

- The client sends no Accept-Encoding request header
- No 'gzip' or 'x-gzip' is present in the Accept-Encoding header
- No 'gzip' or 'x-gzip' with a qvalue > 0 is present
- The 'identity' value is given with a qvalue > 0.

circuits.web.tools.**serve_download**(*request*, *response*, *path*, *name=None*)
Serve 'path' as an application/x-download attachment.

circuits.web.tools.**serve_file**(*request*, *response*, *path*, *type=None*, *disposition=None*, *name=None*)
Set status, headers, and body in order to serve the given file.

The Content-Type header will be set to the type arg, if provided. If not provided, the Content-Type will be guessed by the file extension of the 'path' argument.

If disposition is not None, the Content-Disposition header will be set to "<disposition>; filename=<name>". If name is None, it will be set to the basename of path. If disposition is None, no Content-Disposition header will be written.

circuits.web.tools.**validate_etags**(*request*, *response*, *autotags=False*)
Validate the current ETag against If-Match, If-None-Match headers.

If autotags is True, an ETag response-header value will be provided from an MD5 hash of the response body (unless some other code has already provided an ETag header). If False (the default), the ETag will not be automatic.

WARNING: the autotags feature is not designed for URL's which allow methods other than GET. For example, if a POST to the same URL returns no content, the automatic ETag will be incorrect, breaking a fundamental use for entity tags in a possibly destructive fashion. Likewise, if you raise 304 Not Modified, the response body will be empty, the ETag hash will be incorrect, and your application will break. See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.24

circuits.web.tools.**validate_since**(*request*, *response*)
Validate the current Last-Modified against If-Modified-Since headers.

If no code has set the Last-Modified response header, then no validation will be performed.

**circuits.web.url module** This is a module for dealing with urls. In particular, sanitizing them.

**class** circuits.web.url.**URL**(*scheme*, *host*, *port*, *path*, *params=''*, *query=''*, *fragment=''*)
Bases: object

**For more information on how and what we parse / sanitize:** http://tools.ietf.org/html/rfc1808.html

**The more up-to-date RFC is this one:** http://www.ietf.org/rfc/rfc3986.txt

**absolute**()
Return True if this is a fully-qualified URL with a hostname and everything

**abspath**()
Clear out any '..' and excessive slashes from the path

**canonical**()
Canonicalize this url. This includes reordering parameters and args to have a consistent ordering

**defrag**()
Remove the fragment from this url

**deparam**(*params=None*)
Strip any of the provided parameters out of the url

**encode**(*encoding*)
Return the url in an arbitrary encoding

**equiv**(*other*)
Return true if this url is equivalent to another

**escape**()
Make sure that the path is correctly escaped

**lower**()
Lowercase the hostname

**classmethod parse**(*url*, *encoding*)
Parse the provided url, and return a URL instance

**punycode**()
Convert to punycode hostname

**relative**(*path*, *encoding='utf-8'*)
Evaluate the new path relative to the current url

**sanitize**()
A shortcut to abspath, escape and lowercase

**unescape**()
Unescape the path

**unicode**()
Return a unicode version of this url

**unpunycode**()
Convert to an unpunycoded hostname

**utf8**()
Return a utf-8 version of this url

circuits.web.url.**parse_url**(*url*, *encoding='utf-8'*)
Parse the provided url string and return an URL object

**circuits.web.utils module** Utilities

This module implements utility functions.

circuits.web.utils.**average**(*xs*)

circuits.web.utils.**compress**(*body*, *compress_level*)
Compress 'body' at the given compress_level.

circuits.web.utils.**dictform**(*form*)

circuits.web.utils.**get_ranges**(*headervalue*, *content_length*)
Return a list of (start, stop) indices from a Range header, or None.

Each (start, stop) tuple will be composed of two ints, which are suitable for use in a slicing operation. That is, the header "Range: bytes=3-6", if applied against a Python string, is requesting resource[3:7]. This function will return the list [(3, 7)].

If this function returns an empty list, you should return HTTP 416.

circuits.web.utils.**parse_body**(*request*, *response*, *params*)

circuits.web.utils.**parse_qs**(*query_string*) → dict
Build a params dictionary from a query_string. If keep_blank_values is True (the default), keep values that are blank.

circuits.web.utils.**stddev**(*xs*)

circuits.web.utils.**variance**(*xs*)

**circuits.web.wrappers module** Request/Response Wrappers

This module implements the Request and Response objects.

**class** circuits.web.wrappers.**Body**
Bases: object

Response Body

**class** circuits.web.wrappers.**HTTPStatus**(*status=200*, *reason=None*)
Bases: object

**reason**

**status**

**class** circuits.web.wrappers.**Host**(*ip*, *port*, *name=None*)
Bases: object

An internet address.

name should be the client's host name. If not available (because no DNS lookup is performed), the IP address should be used instead.

**ip = '0.0.0.0'**

**name = 'unknown.tld'**

**port = 80**

**class** circuits.web.wrappers.**Request**(*sock*, *method='GET'*, *scheme='http'*, *path='/'*, *protocol=(1, 1)*, *qs=''*, *headers=None*, *server=None*)
Bases: object

Creates a new Request object to hold information about a request.

**Parameters**

- **sock** (*socket.socket*) – The socket object of the request.
- **method** (*str*) – The requested method.
- **scheme** (*str*) – The requested scheme.
- **path** (*str*) – The requested path.
- **protocol** (*str*) – The requested protocol.

- **qs** (*str*) – The query string of the request.

initializes x; see x.__class__.__doc__ for signature

**handled = False**

**host = ''**

**index = None**

**local = Host('127.0.0.1', 80, '127.0.0.1')**

**login = None**

**protocol = (1, 1)**

**remote = Host('', 0, '')**

**scheme = 'http'**

**script_name = ''**

**server = None**

**class** circuits.web.wrappers.**Response**(*sock*, *request*) → new Response object
Bases: object

A Response object that holds the response to send back to the client. This ensure that the correct data is sent in the correct order.

initializes x; see x.__class__.__doc__ for signature

**body**
Response Body

**chunked = False**

**close = False**

**done = False**

**prepare**()

**status**
Response Status

**stream = False**

**class** circuits.web.wrappers.**Status**
Bases: object

Response Status

circuits.web.wrappers.**file_generator**(*input*, *chunkSize=4096*)

**circuits.web.wsgi module** WSGI Components

This module implements WSGI Components.

**class** circuits.web.wsgi.**Application**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**getRequestResponse**(*environ*)

**headerNames = {'CONTENT_LENGTH': 'Content-Length', 'REMOTE_HOST': 'Remote-Host', 'CONTENT_TYPE':**

**host**

**init**()

**port**

**response**(*event*, *response*)

**secure**

**translateHeaders**(*environ*)

**class** circuits.web.wsgi.**Gateway**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**init**(*apps*)

circuits.web.wsgi.**create_environ**(*errors*, *path*, *req*)

**Module contents** Circuits Library - Web

circuits.web contains the circuits full stack web server that is HTTP and WSGI compliant.

## Submodules

### circuits.six module

Utilities for writing code that runs on Python 2 and 3

circuits.six.**byteindex**(*data*, *index*)

circuits.six.**iterbytes**(*data*)

**class** circuits.six.**MovedModule**(*name*, *old*, *new=None*)
Bases: circuits.six._LazyDescr

**class** circuits.six.**MovedAttribute**(*name*, *old_mod*, *new_mod*, *old_attr=None*, *new_attr=None*)
Bases: circuits.six._LazyDescr

circuits.six.**add_move**(*move*)
Add an item to six.moves.

circuits.six.**remove_move**(*name*)
Remove item from six.moves.

circuits.six.**create_bound_method**(*function*, *instance*)

circuits.six.**get_unbound_function**(*unbound*)
Get the function out of a possibly unbound function

**class** circuits.six.**Iterator**
Bases: object

**next**()

circuits.six.**iterkeys**(*d*)
Return an iterator over the keys of a dictionary.

circuits.six.**itervalues**(*d*)
Return an iterator over the values of a dictionary.

circuits.six.**iteritems**(*d*)
Return an iterator over the (key, value) pairs of a dictionary.

circuits.six.**b**(*s*, *encoding='utf-8'*)
Byte literal

circuits.six.**u**(*s*, *encoding='utf-8'*)
Text literal

circuits.six.**bytes_to_str**(*s*)

circuits.six.**reraise**(*tp*, *value*, *tb=None*)
Reraise an exception.

circuits.six.**exec_**(*code*, *globs=None*, *locs=None*)
Execute code in a namespace.

circuits.six.**print_**(**args*, ***kwargs*)
The new-style print function.

circuits.six.**with_metaclass**(*meta*, *base=<type 'object'>*)
Create a base class with a metaclass.

#### circuits.version module

### Module contents

Lightweight Event driven and Asynchronous Application Framework

circuits is a **Lightweight Event** driven and **Asynchronous Application Framework** for the Python Programming Language with a strong **Component** Architecture.

**copyright** CopyRight (C) 2004-2015 by James Mills

**license** MIT (See: LICENSE)

## 2.5.2 circuits.app package

### Submodules

#### circuits.app.daemon module

Daemon Component

Component to daemonize a system into the background and detach it from its controlling PTY. Supports PID file writing, logging stdin, stdout and stderr and changing the current working directory.

**class** circuits.app.daemon.**Daemon**(**args*, ***kwargs*)
Bases: *circuits.core.components.Component*

Daemon Component

**Parameters**

- **pidfile** (*str or unicode*) – .pid filename
- **stdin** (*str or unicode*) – filename to log stdin

- **stdout** (*str or unicode*) – filename to log stdout
- **stderr** (*str or unicode*) – filename to log stderr

initializes x; see x.__class__.__doc__ for signature

**channel = 'daemon'**

**daemonize**()

**deletepid**()

**init**(*pidfile*, *path='/'*, *stdin=None*, *stdout=None*, *stderr=None*, *channel='daemon'*)

**on_started**(*component*)

**registered**(*component*, *manager*)

**writepid**()

**class** circuits.app.daemon.**daemonize**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

daemonize Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'daemonize'**

**class** circuits.app.daemon.**deletepid**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

"deletepid Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'deletepid'**

**class** circuits.app.daemon.**writepid**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

"writepid Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'writepid'**

## Module contents

Application Components

Contains various components useful for application development and tasks common to applications.

**class** `circuits.app.`**Daemon**(**args*, ***kwargs*)

Bases: *circuits.core.components.Component*

Daemon Component

**Parameters**

- **pidfile** (*str or unicode*) – .pid filename
- **stdin** (*str or unicode*) – filename to log stdin
- **stdout** (*str or unicode*) – filename to log stdout
- **stderr** (*str or unicode*) – filename to log stderr

initializes x; see x.__class__.__doc__ for signature

**channel = 'daemon'**

**daemonize**()

**deletepid**()

**init**(*pidfile*, *path='/'*, *stdin=None*, *stdout=None*, *stderr=None*, *channel='daemon'*)

**on_started**(*component*)

**registered**(*component*, *manager*)

**writepid**()

**class** circuits.app.**DropPrivileges**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**drop_privileges**()

**init**(*user='nobody'*, *group='nobody'*, *umask=63*, ***kwargs*)

**on_ready**(*server*, *bind*)

### 2.5.3 circuits.app.daemon module

Daemon Component

Component to daemonize a system into the background and detach it from its controlling PTY. Supports PID file writing, logging stdin, stdout and stderr and changing the current working directory.

**class** circuits.app.daemon.**Daemon**(**args*, ***kwargs*)
Bases: *circuits.core.components.Component*

Daemon Component

**Parameters**

- **pidfile** (*str or unicode*) – .pid filename
- **stdin** (*str or unicode*) – filename to log stdin
- **stdout** (*str or unicode*) – filename to log stdout
- **stderr** (*str or unicode*) – filename to log stderr

initializes x; see x.__class__.__doc__ for signature

**channel = 'daemon'**

**daemonize**()

**deletepid**()

**init**(*pidfile*, *path='/'*, *stdin=None*, *stdout=None*, *stderr=None*, *channel='daemon'*)

**on_started**(*component*)

**registered**(*component*, *manager*)

**writepid**()

**class** circuits.app.daemon.**daemonize**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

daemonize Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'daemonize'**

**class** `circuits.app.daemon.`**deletepid**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

"deletepid Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'deletepid'**

**class** `circuits.app.daemon.`**writepid**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

"writepid Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'writepid'**

### 2.5.4 circuits.core package

#### Submodules

##### circuits.core.bridge module

Bridge

The Bridge Component is used for inter-process communications between processes. Bridge is used internally when a Component is started in "process mode" via `circuits.core.manager.start()`. Typically a Pipe is used as the socket transport between two sides of a Bridge (*there must be a :class:'~Bridge' instnace on both sides*).

**class** `circuits.core.bridge.`**`ipc`**(*event*, *channel=None*)
Bases: *circuits.core.events.Event*

ipc Event

Send an event to a child/parent process

**Parameters**

- **`event`** (*circuits.core.events.Event*) – Event to execute remotely.
- **`channel`** (*str*) – IPC Channel (channel to use on child/parent).

**`name` = 'ipc'**

**class** `circuits.core.bridge.`**`Bridge`**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**`channel` = 'bridge'**

**`init`**(*socket*, *channel='bridge'*)

##### circuits.core.components module

This module defines the BaseComponent and its subclass Component.

**class** `circuits.core.components.`**`prepare_unregister`**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

This event is fired when a component is about to be unregistered from the component tree. Unregistering a component actually detaches the complete subtree that the unregistered component is the root of. Components that need to know if they are removed from the main tree (e.g. because they maintain relationships to other components in the tree) handle this event, check if the component being unregistered is one of their ancestors and act accordingly.

**Parameters** **`component`** – the component that will be unregistered

**`complete` = True**

**`in_subtree`**(*component*)
Convenience method that checks if the given *component* is in the subtree that is about to be detached.

**`name` = 'prepare_unregister'**

**class** `circuits.core.components.`**`BaseComponent`**(**args*, ***kwargs*)
Bases: *circuits.core.manager.Manager*

This is the base class for all components in a circuits based application. Components can (and should, except for root components) be registered with a parent component.

BaseComponents can declare methods as event handlers using the handler decoration (see *circuits.core.handlers.handler()*). The handlers are invoked for matching events from the component's channel (specified as the component's `channel` attribute).

BaseComponents inherit from *circuits.core.manager.Manager*. This provides components with the *circuits.core.manager.Manager.fireEvent()* method that can be used to fire events as the result of some computation.

Apart from the `fireEvent()` method, the Manager nature is important for root components that are started or run.

**Variables** ***channel*** – a component can be associated with a specific channel by setting this attribute. This should either be done by specifying a class attribute *channel* in the derived class or by passing a keyword parameter *channel="..."* to *__init__*. If specified, the component's handlers receive events on the specified channel only, and events fired by the component will be sent on the specified channel (this behavior may be overridden, see *Event*, *fireEvent()* and *handler()*). By default, the channel attribute is set to "*", meaning that events are fired on all channels and received from all channels.

initializes x; see x.__class__.__doc__ for signature

**channel = '*'**

**register**(*parent*)
Inserts this component in the component tree as a child of the given *parent* node.

**Parameters** **parent** (*Manager*) – the parent component after registration has completed.

This method fires a `Registered` event to inform other components in the tree about the new member.

**unregister**()
Removes this component from the component tree.

Removing a component from the component tree is a two stage process. First, the component is marked as to be removed, which prevents it from receiving further events, and a *prepare_unregister* event is fired. This allows other components to e.g. release references to the component to be removed before it is actually removed from the component tree.

After the processing of the `prepare_unregister` event has completed, the component is removed from the tree and an *unregistered* event is fired.

**unregister_pending**

**classmethod handlers**()
Returns a list of all event handlers for this Component

**classmethod events**()
Returns a list of all events this Component listens to

**classmethod handles**(**names*)
Returns True if all names are event handlers of this Component

**class** circuits.core.components.**Component**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If you use Component instead of BaseComponent as base class for your own component class, then all methods that are not marked as private (i.e: start with an underscore) are automatically decorated as handlers.

The methods are invoked for all events from the component's channel where the event's name matches the method's name.

### circuits.core.debugger module

Debugger component used to debug each event in a system by printing each event to sys.stderr or to a Logger Component instance.

**class** circuits.core.debugger.**Debugger**(*errors=True*, *events=True*, *file=None*, *logger=None*, *prefix=None*, *trim=None*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

Create a new Debugger Component

Creates a new Debugger Component that listens to all events in the system printing each event to sys.stderr or a Logger Component.

**Variables**

- ***IgnoreEvents*** – list of events (str) to ignore
- ***IgnoreChannels*** – list of channels (str) to ignore
- **enabled** – Enabled/Disabled flag

**Parameters** **log** – Logger Component instance or None (*default*)

initializes x; see x.__class__.__doc__ for signature

**IgnoreEvents = ['generate_events']**

**IgnoreChannels = []**

### circuits.core.events module

This module defines the basic event class and common events.

**class** circuits.core.events.**EventType**

Bases: type

**class** circuits.core.events.**Event**(**args*, ***kwargs*)

Bases: object

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**channels = ()**
The channels this message is sent to.

**parent = None**

**notify = False**

**success = False**

**failure = False**

**complete = False**

**alert_done = False**

**waitingHandlers = 0**

**classmethod create**(*name*, **args*, ***kwargs*)

**child**(*name*, **args*, ***kwargs*)

**name = 'Event'**

**cancel**()
Cancel the event from being processed (if not already)

**stop**()
Stop further processing of this event

**class** circuits.core.events.**exception**(*type*, *value*, *traceback*, *handler=None*, *fevent=None*)
Bases: *circuits.core.events.Event*

exception Event

This event is sent for any exceptions that occur during the execution of an event Handler that is not SystemExit or KeyboardInterrupt.

**Parameters**

- **type** (*type*) – type of exception
- **value** (*exceptions.TypeError*) – exception object
- **traceback** (traceback) – traceback of exception
- **handler** (*@handler(<method>)*) – handler that raised the exception
- **fevent** (*event*) – event that failed

**name = 'exception'**

**class** circuits.core.events.**started**(*manager*)
Bases: *circuits.core.events.Event*

started Event

This Event is sent when a Component or Manager has started running.

**Parameters** **manager** (*Component or Manager*) – The component or manager that was started

**name = 'started'**

**class** circuits.core.events.**stopped**(*manager*)
Bases: *circuits.core.events.Event*

stopped Event

This Event is sent when a Component or Manager has stopped running.

**Parameters** **manager** (*Component or Manager*) – The component or manager that has stopped

**name = 'stopped'**

**class** circuits.core.events.**signal**(*signo*, *stack*)
Bases: *circuits.core.events.Event*

signal Event

This Event is sent when a Component receives a signal.

**Parameters**

- **signo** – The signal number received.
- **stack** – The interrupted stack frame.

**name = 'signal'**

**class** circuits.core.events.**registered**(*component*, *manager*)
Bases: *circuits.core.events.Event*

registered Event

This Event is sent when a Component has registered with another Component or Manager. This Event is only sent if the Component or Manager being registered which is not itself.

**Parameters**

- **component** (Component) – The Component being registered
- **manager** (*Component or Manager*) – The Component or Manager being registered with

**name = 'registered'**

**class** circuits.core.events.**unregistered**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

unregistered Event

This Event is sent when a Component has been unregistered from its Component or Manager.

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'unregistered'**

**class** `circuits.core.events.`**generate_events**(*lock*, *max_wait*)
Bases: *circuits.core.events.Event*

generate_events Event

This Event is sent by the circuits core. All components that generate timed events or events from external sources (e.g. data becoming available) should fire any pending events in their "generate_events" handler.

The handler must either call `stop()` (*preventing other handlers from being called in the same iteration) or must invoke *reduce_time_left()* with parameter 0.

**Parameters** **max_wait** – maximum time available for generating events.

Components that actually consume time waiting for events to be generated, thus suspending normal execution, must provide a method `resume` that interrupts waiting for events.

**time_left**
The time left for generating events. A value less than 0 indicates unlimited time. You should have only one component in your system (usually a poller component) that spends up to "time left" until it generates an event.

**reduce_time_left**(*time_left*)
Update the time left for generating events. This is typically used by event generators that currently don't want to generate an event but know that they will within a certain time. By reducing the time left, they make sure that they are reinvoked when the time for generating the event has come (at the latest).

This method can only be used to reduce the time left. If the parameter is larger than the current value of time left, it is ignored.

If the time left is reduced to 0 and the event is currently being handled, the handler's *resume* method is invoked.

**lock**

**name = 'generate_events'**

### circuits.core.handlers module

This module define the @handler decorator/function and the HandlesType type.

circuits.core.handlers.**handler**(**names*, ***kwargs*)

Creates an Event Handler

This decorator can be applied to methods of classes derived from *circuits.core.components.BaseComponent*. It marks the method as a handler for the events passed as arguments to the `@handler` decorator. The events are specified by their name.

The decorated method's arguments must match the arguments passed to the *circuits.core.events.Event* on creation. Optionally, the method may have an additional first argument named *event*. If declared, the event object that caused the handler to be invoked is assigned to it.

By default, the handler is invoked by the component's root *Manager* for events that are propagated on the channel determined by the BaseComponent's *channel* attribute. This may be overridden by specifying a different channel as a keyword parameter of the decorator (`channel=...`).

Keyword argument `priority` influences the order in which handlers for a specific event are invoked. The higher the priority, the earlier the handler is executed.

If you want to override a handler defined in a base class of your component, you must specify `override=True`, else your method becomes an additional handler for the event.

**Return value**

Normally, the results returned by the handlers for an event are simply collected in the *circuits.core.events.Event*'s `value` attribute. As a special case, a handler may return a `types.GeneratorType`. This signals to the dispatcher that the handler isn't ready to deliver a result yet. Rather, it has interrupted it's execution with a `yield None` statement, thus preserving its current execution state.

The dispatcher saves the returned generator object as a task. All tasks are reexamined (i.e. their `next()` method is invoked) when the pending events have been executed.

This feature avoids an unnecessarily complicated chaining of event handlers. Imagine a handler A that needs the results from firing an event E in order to complete. Then without this feature, the final action of A would be to fire event E, and another handler for an event `SuccessE` would be required to complete handler A's operation, now having the result from invoking E available (actually it's even a bit more complicated).

Using this "suspend" feature, the handler simply fires event E and then yields `None` until e.g. it finds a result in E's `value` attribute. For the simplest scenario, there even is a utility method *circuits.core.manager.Manager.callEvent()* that combines firing and waiting.

**class** circuits.core.handlers.**Unknown**

Bases: `object`

Unknown Dummy Component

circuits.core.handlers.**reprhandler**(*handler*)

**class** circuits.core.handlers.**HandlerMetaClass**(*name*, *bases*, *ns*)

Bases: `type`

### circuits.core.helpers module

**class** circuits.core.helpers.**FallBackGenerator**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**resume**()
Implements the resume method as required from components that handle GenerateEvents.

**class** circuits.core.helpers.**FallBackExceptionHandler**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If there is no handler for error events in the component hierarchy, this component's handler is added automatically. It simply prints the error information on stderr.

initializes x; see x.__class__.__doc__ for signature

**class** circuits.core.helpers.**FallBackSignalHandler**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If there is no handler for signal events in the component hierarchy, this component's handler is added automatically. It simply terminates the system if the signal is SIGINT or SIGTERM.

initializes x; see x.__class__.__doc__ for signature

### circuits.core.loader module

This module implements a generic Loader suitable for dynamically loading components from other modules. This supports loading from local paths, eggs and zip archives. Both setuptools and distribute are fully supported.

**class** circuits.core.loader.**Loader**(*auto_register=True*, *init_args=None*, *init_kwargs=None*, *paths=None*, *channel='loader'*)
Bases: *circuits.core.components.BaseComponent*

Create a new Loader Component

Creates a new Loader Component that enables dynamic loading of components from modules either in local paths, eggs or zip archives.

initializes x; see x.__class__.__doc__ for signature

**channel = 'loader'**

**load**(*name*)

### circuits.core.manager module

This module defines the Manager class.

**exception** circuits.core.manager.**UnregistrableError**
Bases: exceptions.Exception

Raised if a component cannot be registered as child.

**exception** circuits.core.manager.**TimeoutError**
Bases: exceptions.Exception

Raised if wait event timeout occurred

**class** circuits.core.manager.**CallValue**(*value*)
Bases: object

**class** circuits.core.manager.**ExceptionWrapper**(*exception*)
Bases: object

**extract**()

**class** circuits.core.manager.**Sleep**(*seconds*)
Bases: *circuits.six.Iterator*

**expired**

**task**

circuits.core.manager.**sleep**(*seconds*)
Delay execution of a coroutine for a given number of seconds. The argument may be a floating point number for subsecond precision.

**class** circuits.core.manager.**Manager**(**args*, ***kwargs*)
Bases: object

The manager class has two roles. As a base class for component implementation, it provides methods for event and handler management. The method *fireEvent()* appends a new event at the end of the event queue for later execution. *waitEvent()* suspends the execution of a handler until all handlers for a given event have been invoked. *callEvent()* combines the last two methods in a single method.

The methods *addHandler()* and *removeHandler()* allow handlers for events to be added and removed dynamically. (The more common way to register a handler is to use the *handler()* decorator or derive the class from *Component*.)

In its second role, the *Manager* takes the role of the event executor. Every component hierarchy has a root component that maintains a queue of events. Firing an event effectively means appending it to the event queue maintained by the root manager. The *flush()* method removes all pending events from the queue and, for each event, invokes all the handlers. Usually, *flush()* is indirectly invoked by *run()*.

The manager optionally provides information about the execution of events as automatically generated events. If an *Event* has its success attribute set to True, the manager fires a Success event if all handlers have been executed without error. Note that this event will be enqueued (and dispatched) immediately after the events that have been fired by the event's handlers. So the success event indicates both the successful invocation of all handlers for the event and the processing of the immediate follow-up events fired by those handlers.

Sometimes it is not sufficient to know that an event and its immediate follow-up events have been processed. Rather, it is important to know when all state changes triggered by an event, directly or indirectly, have been performed. This also includes the processing of events that have been fired when invoking the handlers for the follow-up events and the processing of events that have again been fired by those handlers and so on. The completion of the processing of an event and all its direct or indirect follow-up events may be indicated by a Complete event. This event is generated by the manager if *Event* has its complete attribute set to True.

Apart from the event queue, the root manager also maintains a list of tasks, actually Python generators, that are updated when the event queue has been flushed.

initializes x; see x.__class__.__doc__ for signature

**name**
Return the name of this Component/Manager

**running**
Return the running state of this Component/Manager

**pid**
Return the process id of this Component/Manager

**getHandlers**(*event*, *channel*, ***kwargs*)

**addHandler**(*f*)

**removeHandler**(*method*, *event=None*)

**registerChild**(*component*)

**unregisterChild**(*component*)

**fireEvent**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**fire**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**registerTask**(*g*)

**unregisterTask**(*g*)

**waitEvent**(*event*, **channels*, ***kwargs*)

**wait**(*event*, **channels*, ***kwargs*)

**callEvent**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**call**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**flushEvents**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**flush**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**start**(*process=False*, *link=None*)
Start a new thread or process that invokes this manager's `run()` method. The invocation of this method returns immediately after the task or process has been started.

**join**()

**stop**(*code=None*)
Stop this manager. Invoking this method causes an invocation of `run()` to return.

**processTask**(*event*, *task*, *parent=None*)

**tick**(*timeout=-1*)
Execute all possible actions once. Process all registered tasks and flush the event queue. If the application is running fire a GenerateEvents to get new events from sources.

This method is usually invoked from *run()*. It may also be used to build an application specific main loop.

**Parameters** **timeout** (*float, measuring seconds*) – the maximum waiting time spent in this method. If negative, the method may block until at least one action has been taken.

**run**(*socket=None*)
Run this manager. The method fires the `Started` event and then continuously calls *tick()*.

The method returns when the manager's *stop()* method is invoked.

If invoked by a programs main thread, a signal handler for the `INT` and `TERM` signals is installed. This handler fires the corresponding `Signal` events and then calls *stop()* for the manager.

### circuits.core.pollers module

Poller Components for asynchronous file and socket I/O.

This module contains Poller components that enable polling of file or socket descriptors for read/write events. Pollers: - Select - Poll - EPoll

**class** circuits.core.pollers.**BasePoller**(*channel=None*)
Bases: *circuits.core.components.BaseComponent*

**channel** = **None**

**resume**()

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**isReading**(*fd*)

**isWriting**(*fd*)

**discard**(*fd*)

**getTarget**(*fd*)

**class** circuits.core.pollers.**Select**(...) → new Select Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new Select Poller Component that uses the select poller implementation. This poller is not recommended but is available for legacy reasons as most systems implement select-based polling for backwards compatibility.

**channel** = **'select'**

**class** circuits.core.pollers.**Poll**(...) → new Poll Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new Poll Poller Component that uses the poll poller implementation.

**channel = 'poll'**

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**discard**(*fd*)

**class** circuits.core.pollers.**EPoll**(...) → new EPoll Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new EPoll Poller Component that uses the epoll poller implementation.

**channel = 'epoll'**

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**discard**(*fd*)

**class** circuits.core.pollers.**KQueue**(...) → new KQueue Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new KQueue Poller Component that uses the kqueue poller implementation.

**channel = 'kqueue'**

**addReader**(*source*, *sock*)

**addWriter**(*source*, *sock*)

**removeReader**(*sock*)

**removeWriter**(*sock*)

**discard**(*sock*)

circuits.core.pollers.**Poller**
alias of *Select*

### circuits.core.timers module

Timer component to facilitate timed events.

**class** circuits.core.timers.**Timer**(*interval*, *event*, **channels*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Timer Component

A timer is a component that fires an event once after a certain delay or periodically at a regular interval.

**Parameters**

- **interval** (`datetime` or number of seconds as a `float`) – the delay or interval to wait for until the event is fired. If interval is specified as datetime, the interval is recalculated as the time span from now to the given datetime.
- **event** (*Event*) – the event to fire.
- **persist** (`bool`) – An optional keyword argument which if `True` will cause the event to be fired repeatedly once per configured interval until the timer is unregistered. **Default:** `False`

**reset** (*interval=None*)
Reset the timer, i.e. clear the amount of time already waited for.

**expiry**

### circuits.core.utils module

Utils

This module defines utilities used by circuits.

`circuits.core.utils.`**flatten** (*root*, *visited=None*)

`circuits.core.utils.`**findchannel** (*root*, *channel*, *all=False*)

`circuits.core.utils.`**findtype** (*root*, *component*, *all=False*)

`circuits.core.utils.`**findcmp** (*root*, *component*, *all=False*)

`circuits.core.utils.`**findroot** (*component*)

`circuits.core.utils.`**safeimport** (*name*)

### circuits.core.values module

This defines the Value object used by components and events.

**class** `circuits.core.values.`**Value** (*event=None*, *manager=None*)
Bases: `object`

Create a new future Value Object

Creates a new future Value Object which is used by Event Objects and the Manager to store the result(s) of an Event Handler's exeuction of some Event in the system.

**Parameters**

- **event** (*Event instance*) – The Event this Value is associated with.
- **manager** (*A Manager/Component instance.*) – The Manager/Component used to trigger notifications.

**Variables**

- **result** – True if this value has been changed.
- ***errors*** – True if while setting this value an exception occured.
- ***notify*** – True or an event name to notify of changes to this value

This is a Future/Promise implementation.

**inform** (*force=False*)

**getValue**(*recursive=True*)

**setValue**(*value*)

**value**
Value of this Value

### circuits.core.workers module

Workers

Worker components used to perform "work" in independent threads or processes. Worker(s) are typically used by a Pool (circuits.core.pools) to create a pool of workers. Worker(s) are not registered with a Manager or another Component - instead they are managed by the Pool. If a Worker is used independently it should not be registered as it causes its main event handler `_on_task` to execute in the other thread blocking it.

**class** circuits.core.workers.**task**(*f*, **args*, ***kwargs*)
Bases: *circuits.core.events.Event*

task Event

This Event is used to initiate a new task to be performed by a Worker or a Pool of Worker(s).

**Parameters**

- **f** (*function*) – The function to be executed.
- **args** (*tuple*) – Arguments to pass to the function
- **kwargs** (*dict*) – Keyword Arguments to pass to the function

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**success = True**

**failure = True**

**name = 'task'**

**class** circuits.core.workers.**Worker**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

A thread/process Worker Component

This Component creates a Worker (either a thread or process) which when given a `Task`, will execute the given function in the task in the background in its thread/process.

**Parameters** **process** (*bool*) – True to start this Worker as a process (Thread otherwise)

initializes x; see x.__class__.__doc__ for signature

**channel = 'worker'**

**init**(*process=False*, *workers=None*, *channel='worker'*)

## Module contents

Core

This package contains the essential core parts of the circuits framework.

circuits.core.**handler**(**names*, ***kwargs*)

Creates an Event Handler

This decorator can be applied to methods of classes derived from *circuits.core.components.BaseComponent*. It marks the method as a handler for the events passed as arguments to the `@handler` decorator. The events are specified by their name.

The decorated method's arguments must match the arguments passed to the *circuits.core.events.Event* on creation. Optionally, the method may have an additional first argument named *event*. If declared, the event object that caused the handler to be invoked is assigned to it.

By default, the handler is invoked by the component's root *Manager* for events that are propagated on the channel determined by the BaseComponent's *channel* attribute. This may be overridden by specifying a different channel as a keyword parameter of the decorator (`channel=...`).

Keyword argument `priority` influences the order in which handlers for a specific event are invoked. The higher the priority, the earlier the handler is executed.

If you want to override a handler defined in a base class of your component, you must specify `override=True`, else your method becomes an additional handler for the event.

**Return value**

Normally, the results returned by the handlers for an event are simply collected in the *circuits.core.events.Event*'s `value` attribute. As a special case, a handler may return a `types.GeneratorType`. This signals to the dispatcher that the handler isn't ready to deliver a result yet. Rather, it has interrupted it's execution with a `yield None` statement, thus preserving its current execution state.

The dispatcher saves the returned generator object as a task. All tasks are reexamined (i.e. their `next()` method is invoked) when the pending events have been executed.

This feature avoids an unnecessarily complicated chaining of event handlers. Imagine a handler A that needs the results from firing an event E in order to complete. Then without this feature, the final action of A would be to fire event E, and another handler for an event `SuccessE` would be required to complete handler A's operation, now having the result from invoking E available (actually it's even a bit more complicated).

Using this "suspend" feature, the handler simply fires event E and then yields `None` until e.g. it finds a result in E's `value` attribute. For the simplest scenario, there even is a utility method *circuits.core.manager.Manager.callEvent()* that combines firing and waiting.

**class** circuits.core.**BaseComponent**(**args*, ***kwargs*)

Bases: *circuits.core.manager.Manager*

This is the base class for all components in a circuits based application. Components can (and should, except for root components) be registered with a parent component.

BaseComponents can declare methods as event handlers using the handler decoration (see *circuits.core.handlers.handler()*). The handlers are invoked for matching events from the component's channel (specified as the component's `channel` attribute).

BaseComponents inherit from *circuits.core.manager.Manager*. This provides components with the *circuits.core.manager.Manager.fireEvent()* method that can be used to fire events as the result of some computation.

Apart from the `fireEvent()` method, the Manager nature is important for root components that are started or run.

**Variables** ***channel*** – a component can be associated with a specific channel by setting this attribute. This should either be done by specifying a class attribute *channel* in the derived class or by passing a keyword parameter *channel="..."* to *__init__*. If specified, the component's handlers receive events on the specified channel only, and events fired by the component will

be sent on the specified channel (this behavior may be overridden, see *Event*, *fireEvent()* and *handler()*). By default, the channel attribute is set to "*", meaning that events are fired on all channels and received from all channels.

initializes x; see x.__class__.__doc__ for signature

**channel = '*'**

**classmethod events()**
Returns a list of all events this Component listens to

**classmethod handlers()**
Returns a list of all event handlers for this Component

**classmethod handles**(**names*)
Returns True if all names are event handlers of this Component

**register**(*parent*)
Inserts this component in the component tree as a child of the given *parent* node.

**Parameters** **parent** (*Manager*) – the parent component after registration has completed.

This method fires a `Registered` event to inform other components in the tree about the new member.

**unregister()**
Removes this component from the component tree.

Removing a component from the component tree is a two stage process. First, the component is marked as to be removed, which prevents it from receiving further events, and a *prepare_unregister* event is fired. This allows other components to e.g. release references to the component to be removed before it is actually removed from the component tree.

After the processing of the `prepare_unregister` event has completed, the component is removed from the tree and an *unregistered* event is fired.

**unregister_pending**

**class** circuits.core.**Component**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**class** circuits.core.**Event**(**args*, ***kwargs*)
Bases: object

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**alert_done = False**

**cancel**()
Cancel the event from being processed (if not already)

**channels = ()**

**child**(*name*, **args*, ***kwargs*)

**complete = False**

**classmethod create**(*name*, **args*, ***kwargs*)

**failure = False**

**name = 'Event'**

**notify = False**

**parent = None**

**stop**()
Stop further processing of this event

**success = False**

**waitingHandlers = 0**

**class** circuits.core.**task**(*f*, **args*, ***kwargs*)
Bases: *circuits.core.events.Event*

task Event

This Event is used to initiate a new task to be performed by a Worker or a Pool of Worker(s).

**Parameters**

- **f** (*function*) – The function to be executed.
- **args** (*tuple*) – Arguments to pass to the function
- **kwargs** (*dict*) – Keyword Arguments to pass to the function

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**failure = True**

**name = 'task'**

**success = True**

**class** circuits.core.**Worker**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

A thread/process Worker Component

This Component creates a Worker (either a thread or process) which when given a `Task`, will execute the given function in the task in the background in its thread/process.

**Parameters** **process** (*bool*) – True to start this Worker as a process (Thread otherwise)

initializes x; see x.__class__.__doc__ for signature

**channel = 'worker'**

**init**(*process=False*, *workers=None*, *channel='worker'*)

**class** circuits.core.**ipc**(*event*, *channel=None*)
Bases: *circuits.core.events.Event*

ipc Event

Send an event to a child/parent process

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **channel** (*str*) – IPC Channel (channel to use on child/parent).

**name = 'ipc'**

**class** circuits.core.**Bridge**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'bridge'**

**init**(*socket*, *channel='bridge'*)

**class** circuits.core.**Debugger**(*errors=True*, *events=True*, *file=None*, *logger=None*, *prefix=None*, *trim=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Create a new Debugger Component

Creates a new Debugger Component that listens to all events in the system printing each event to sys.stderr or a Logger Component.

**Variables**

- ***IgnoreEvents*** – list of events (str) to ignore
- ***IgnoreChannels*** – list of channels (str) to ignore
- **enabled** – Enabled/Disabled flag

**Parameters** **log** – Logger Component instance or None (*default*)

initializes x; see x.__class__.__doc__ for signature

**IgnoreChannels = []**

**IgnoreEvents = ['generate_events']**

**class** circuits.core.**Timer**(*interval*, *event*, **channels*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Timer Component

A timer is a component that fires an event once after a certain delay or periodically at a regular interval.

**Parameters**

- **interval** (datetime or number of seconds as a float) – the delay or interval to wait for until the event is fired. If interval is specified as datetime, the interval is recalculated as the time span from now to the given datetime.
- **event** (*Event*) – the event to fire.
- **persist** (bool) – An optional keyword argument which if True will cause the event to be fired repeatedly once per configured interval until the timer is unregistered. **Default:** False

**expiry**

**reset**(*interval=None*)
Reset the timer, i.e. clear the amount of time already waited for.

**class** circuits.core.**Manager**(**args*, ***kwargs*)
Bases: object

The manager class has two roles. As a base class for component implementation, it provides methods for event and handler management. The method *fireEvent()* appends a new event at the end of the event queue for later execution. *waitEvent()* suspends the execution of a handler until all handlers for a given event have been invoked. *callEvent()* combines the last two methods in a single method.

The methods *addHandler()* and *removeHandler()* allow handlers for events to be added and removed dynamically. (The more common way to register a handler is to use the *handler()* decorator or derive the class from *Component*.)

In its second role, the *Manager* takes the role of the event executor. Every component hierarchy has a root component that maintains a queue of events. Firing an event effectively means appending it to the event queue maintained by the root manager. The *flush()* method removes all pending events from the queue and, for each event, invokes all the handlers. Usually, *flush()* is indirectly invoked by *run()*.

The manager optionally provides information about the execution of events as automatically generated events. If an *Event* has its success attribute set to True, the manager fires a Success event if all handlers have been executed without error. Note that this event will be enqueued (and dispatched) immediately after the events that have been fired by the event's handlers. So the success event indicates both the successful invocation of all handlers for the event and the processing of the immediate follow-up events fired by those handlers.

Sometimes it is not sufficient to know that an event and its immediate follow-up events have been processed. Rather, it is important to know when all state changes triggered by an event, directly or indirectly, have been performed. This also includes the processing of events that have been fired when invoking the handlers for the follow-up events and the processing of events that have again been fired by those handlers and so on. The completion of the processing of an event and all its direct or indirect follow-up events may be indicated by a Complete event. This event is generated by the manager if *Event* has its complete attribute set to True.

Apart from the event queue, the root manager also maintains a list of tasks, actually Python generators, that are updated when the event queue has been flushed.

initializes x; see x.__class__.__doc__ for signature

**addHandler**(*f*)

**call**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This

method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**callEvent**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**fire**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**fireEvent**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**flush**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**flushEvents**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**getHandlers**(*event*, *channel*, ***kwargs*)

**join**()

**name**
Return the name of this Component/Manager

**pid**
Return the process id of this Component/Manager

**processTask**(*event*, *task*, *parent=None*)

**registerChild**(*component*)

**registerTask**(*g*)

**removeHandler**(*method*, *event=None*)

**run**(*socket=None*)
Run this manager. The method fires the `Started` event and then continuously calls *tick()*.

The method returns when the manager's *stop()* method is invoked.

If invoked by a programs main thread, a signal handler for the `INT` and `TERM` signals is installed. This handler fires the corresponding `Signal` events and then calls *`stop()`* for the manager.

**`running`**
Return the running state of this Component/Manager

**`start`**(*process=False*, *link=None*)
Start a new thread or process that invokes this manager's `run()` method. The invocation of this method returns immediately after the task or process has been started.

**`stop`**(*code=None*)
Stop this manager. Invoking this method causes an invocation of `run()` to return.

**`tick`**(*timeout=-1*)
Execute all possible actions once. Process all registered tasks and flush the event queue. If the application is running fire a GenerateEvents to get new events from sources.

This method is usually invoked from *`run()`*. It may also be used to build an application specific main loop.

**Parameters** **`timeout`** (*float, measuring seconds*) – the maximum waiting time spent in this method. If negative, the method may block until at least one action has been taken.

**`unregisterChild`**(*component*)

**`unregisterTask`**(*g*)

**`wait`**(*event*, **channels*, ***kwargs*)

**`waitEvent`**(*event*, **channels*, ***kwargs*)

**exception** `circuits.core.`**`TimeoutError`**
Bases: `exceptions.Exception`

Raised if wait event timeout occurred

### 2.5.5 circuits.core.bridge module

Bridge

The Bridge Component is used for inter-process communications between processes. Bridge is used internally when a Component is started in "process mode" via `circuits.core.manager.start()`. Typically a Pipe is used as the socket transport between two sides of a Bridge (*there must be a :class:'~Bridge' instnace on both sides*).

**class** `circuits.core.bridge.`**`ipc`**(*event*, *channel=None*)
Bases: *`circuits.core.events.Event`*

ipc Event

Send an event to a child/parent process

**Parameters**

- **`event`** (*`circuits.core.events.Event`*) – Event to execute remotely.
- **`channel`** (*str*) – IPC Channel (channel to use on child/parent).

**`name = 'ipc'`**

**class** `circuits.core.bridge.`**`Bridge`**(**args*, ***kwargs*)
Bases: *`circuits.core.components.BaseComponent`*

initializes x; see x.__class__.__doc__ for signature

**`channel = 'bridge'`**

**init**(*socket*, *channel='bridge'*)

### 2.5.6 circuits.core.components module

This module defines the BaseComponent and its subclass Component.

**class** circuits.core.components.**prepare_unregister**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

This event is fired when a component is about to be unregistered from the component tree. Unregistering a component actually detaches the complete subtree that the unregistered component is the root of. Components that need to know if they are removed from the main tree (e.g. because they maintain relationships to other components in the tree) handle this event, check if the component being unregistered is one of their ancestors and act accordingly.

**Parameters** **component** – the component that will be unregistered

**complete = True**

**in_subtree**(*component*)
Convenience method that checks if the given *component* is in the subtree that is about to be detached.

**name = 'prepare_unregister'**

**class** circuits.core.components.**BaseComponent**(**args*, ***kwargs*)
Bases: *circuits.core.manager.Manager*

This is the base class for all components in a circuits based application. Components can (and should, except for root components) be registered with a parent component.

BaseComponents can declare methods as event handlers using the handler decoration (see *circuits.core.handlers.handler()*). The handlers are invoked for matching events from the component's channel (specified as the component's `channel` attribute).

BaseComponents inherit from *circuits.core.manager.Manager*. This provides components with the *circuits.core.manager.Manager.fireEvent()* method that can be used to fire events as the result of some computation.

Apart from the `fireEvent()` method, the Manager nature is important for root components that are started or run.

**Variables** ***channel*** – a component can be associated with a specific channel by setting this attribute. This should either be done by specifying a class attribute *channel* in the derived class or by passing a keyword parameter *channel="..."* to *__init__*. If specified, the component's handlers receive events on the specified channel only, and events fired by the component will be sent on the specified channel (this behavior may be overridden, see *Event*, *fireEvent()* and *handler()*). By default, the channel attribute is set to "*", meaning that events are fired on all channels and received from all channels.

initializes x; see x.__class__.__doc__ for signature

**channel = '*'**

**register**(*parent*)
Inserts this component in the component tree as a child of the given *parent* node.

**Parameters** **parent** (*Manager*) – the parent component after registration has completed.

This method fires a `Registered` event to inform other components in the tree about the new member.

**unregister**()
Removes this component from the component tree.

Removing a component from the component tree is a two stage process. First, the component is marked as to be removed, which prevents it from receiving further events, and a *prepare_unregister* event is fired. This allows other components to e.g. release references to the component to be removed before it is actually removed from the component tree.

After the processing of the prepare_unregister event has completed, the component is removed from the tree and an *unregistered* event is fired.

**unregister_pending**

**classmethod handlers**()
Returns a list of all event handlers for this Component

**classmethod events**()
Returns a list of all events this Component listens to

**classmethod handles**(**names*)
Returns True if all names are event handlers of this Component

**class** circuits.core.components.**Component**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If you use Component instead of BaseComponent as base class for your own component class, then all methods that are not marked as private (i.e: start with an underscore) are automatically decorated as handlers.

The methods are invoked for all events from the component's channel where the event's name matches the method's name.

### 2.5.7 circuits.core.debugger module

Debugger component used to debug each event in a system by printing each event to sys.stderr or to a Logger Component instance.

**class** circuits.core.debugger.**Debugger**(*errors=True*, *events=True*, *file=None*, *logger=None*, *prefix=None*, *trim=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Create a new Debugger Component

Creates a new Debugger Component that listens to all events in the system printing each event to sys.stderr or a Logger Component.

**Variables**

- ***IgnoreEvents*** – list of events (str) to ignore
- ***IgnoreChannels*** – list of channels (str) to ignore
- **enabled** – Enabled/Disabled flag

**Parameters** **log** – Logger Component instance or None (*default*)

initializes x; see x.__class__.__doc__ for signature

**IgnoreEvents = ['generate_events']**

**IgnoreChannels = []**

### 2.5.8 circuits.core.events module

This module defines the basic event class and common events.

**class** circuits.core.events.**EventType**
Bases: type

**class** circuits.core.events.**Event**(**args*, ***kwargs*)
Bases: object

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to True, an associated event success (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to True, an associated event complete (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**channels = ()**
The channels this message is sent to.

**parent = None**

**notify = False**

**success = False**

**failure = False**

**complete = False**

**alert_done = False**

**waitingHandlers = 0**

**classmethod create**(*name*, **args*, ***kwargs*)

**child**(*name*, **args*, ***kwargs*)

**name = 'Event'**

**cancel**()
Cancel the event from being processed (if not already)

**stop**()
Stop further processing of this event

**class** circuits.core.events.**exception**(*type*, *value*, *traceback*, *handler=None*, *fevent=None*)
Bases: *circuits.core.events.Event*

exception Event

This event is sent for any exceptions that occur during the execution of an event Handler that is not SystemExit or KeyboardInterrupt.

**Parameters**

- **type** (*type*) – type of exception
- **value** (*exceptions.TypeError*) – exception object
- **traceback** (traceback) – traceback of exception
- **handler** (*@handler(<method>)*) – handler that raised the exception
- **fevent** (*event*) – event that failed

**name = 'exception'**

**class** circuits.core.events.**started**(*manager*)
Bases: *circuits.core.events.Event*

started Event

This Event is sent when a Component or Manager has started running.

**Parameters** **manager** (*Component or Manager*) – The component or manager that was started

**name = 'started'**

**class** circuits.core.events.**stopped**(*manager*)
Bases: *circuits.core.events.Event*

stopped Event

This Event is sent when a Component or Manager has stopped running.

**Parameters** **manager** (*Component or Manager*) – The component or manager that has stopped

**name = 'stopped'**

**class** circuits.core.events.**signal**(*signo*, *stack*)
Bases: *circuits.core.events.Event*

signal Event

This Event is sent when a Component receives a signal.

**Parameters**

- **signo** – The signal number received.
- **stack** – The interrupted stack frame.

**name = 'signal'**

**class** circuits.core.events.**registered**(*component*, *manager*)
Bases: *circuits.core.events.Event*

registered Event

This Event is sent when a Component has registered with another Component or Manager. This Event is only sent if the Component or Manager being registered which is not itself.

**Parameters**

- **component** (Component) – The Component being registered
- **manager** (*Component or Manager*) – The Component or Manager being registered with

**name = 'registered'**

**class** circuits.core.events.**unregistered**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

unregistered Event

This Event is sent when a Component has been unregistered from its Component or Manager.

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'unregistered'**

**class** circuits.core.events.**generate_events**(*lock*, *max_wait*)
Bases: *circuits.core.events.Event*

generate_events Event

This Event is sent by the circuits core. All components that generate timed events or events from external sources (e.g. data becoming available) should fire any pending events in their "generate_events" handler.

The handler must either call `stop()` (*preventing other handlers from being called in the same iteration) or must invoke *reduce_time_left()* with parameter 0.

**Parameters** **max_wait** – maximum time available for generating events.

Components that actually consume time waiting for events to be generated, thus suspending normal execution, must provide a method `resume` that interrupts waiting for events.

**time_left**
The time left for generating events. A value less than 0 indicates unlimited time. You should have only one component in your system (usually a poller component) that spends up to "time left" until it generates an event.

**reduce_time_left**(*time_left*)
Update the time left for generating events. This is typically used by event generators that currently don't want to generate an event but know that they will within a certain time. By reducing the time left, they make sure that they are reinvoked when the time for generating the event has come (at the latest).

This method can only be used to reduce the time left. If the parameter is larger than the current value of time left, it is ignored.

If the time left is reduced to 0 and the event is currently being handled, the handler's *resume* method is invoked.

**lock**

**name = 'generate_events'**

### 2.5.9 circuits.core.handlers module

This module define the @handler decorator/function and the HandlesType type.

circuits.core.handlers.**handler**(**names*, ***kwargs*)
Creates an Event Handler

This decorator can be applied to methods of classes derived from *circuits.core.components.BaseComponent*. It marks the method as a handler for the events passed as arguments to the `@handler` decorator. The events are specified by their name.

The decorated method's arguments must match the arguments passed to the *circuits.core.events.Event* on creation. Optionally, the method may have an additional first argument named *event*. If declared, the event object that caused the handler to be invoked is assigned to it.

By default, the handler is invoked by the component's root *Manager* for events that are propagated on the channel determined by the BaseComponent's *channel* attribute. This may be overridden by specifying a different channel as a keyword parameter of the decorator (`channel=...`).

Keyword argument `priority` influences the order in which handlers for a specific event are invoked. The higher the priority, the earlier the handler is executed.

If you want to override a handler defined in a base class of your component, you must specify `override=True`, else your method becomes an additional handler for the event.

**Return value**

Normally, the results returned by the handlers for an event are simply collected in the *circuits.core.events.Event*'s `value` attribute. As a special case, a handler may return a `types.GeneratorType`. This signals to the dispatcher that the handler isn't ready to deliver a result yet. Rather, it has interrupted it's execution with a `yield None` statement, thus preserving its current execution state.

The dispatcher saves the returned generator object as a task. All tasks are reexamined (i.e. their `next()` method is invoked) when the pending events have been executed.

This feature avoids an unnecessarily complicated chaining of event handlers. Imagine a handler A that needs the results from firing an event E in order to complete. Then without this feature, the final action of A would be to fire event E, and another handler for an event `SuccessE` would be required to complete handler A's operation, now having the result from invoking E available (actually it's even a bit more complicated).

Using this "suspend" feature, the handler simply fires event E and then yields `None` until e.g. it finds a result in E's `value` attribute. For the simplest scenario, there even is a utility method *circuits.core.manager.Manager.callEvent()* that combines firing and waiting.

**class** `circuits.core.handlers.`**`Unknown`**
Bases: `object`

Unknown Dummy Component

`circuits.core.handlers.`**`reprhandler`**(*handler*)

**class** `circuits.core.handlers.`**`HandlerMetaClass`**(*name*, *bases*, *ns*)
Bases: `type`

### 2.5.10 circuits.core.helpers module

**class** `circuits.core.helpers.`**`FallBackGenerator`**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**`resume`**()
Implements the resume method as required from components that handle `GenerateEvents`.

**class** `circuits.core.helpers.`**`FallBackExceptionHandler`**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If there is no handler for error events in the component hierarchy, this component's handler is added automatically. It simply prints the error information on stderr.

initializes x; see x.__class__.__doc__ for signature

**class** `circuits.core.helpers.`**`FallBackSignalHandler`**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

If there is no handler for signal events in the component hierarchy, this component's handler is added automatically. It simply terminates the system if the signal is SIGINT or SIGTERM.

initializes x; see x.__class__.__doc__ for signature

### 2.5.11 circuits.core.loader module

This module implements a generic Loader suitable for dynamically loading components from other modules. This supports loading from local paths, eggs and zip archives. Both setuptools and distribute are fully supported.

**class** `circuits.core.loader.`**`Loader`**(*auto_register=True*, *init_args=None*, *init_kwargs=None*, *paths=None*, *channel='loader'*)
Bases: *circuits.core.components.BaseComponent*

Create a new Loader Component

Creates a new Loader Component that enables dynamic loading of components from modules either in local paths, eggs or zip archives.

initializes x; see x.__class__.__doc__ for signature

**channel = 'loader'**

**load**(*name*)

### 2.5.12 circuits.core.manager module

This module defines the Manager class.

**exception** circuits.core.manager.**UnregistrableError**
Bases: exceptions.Exception

Raised if a component cannot be registered as child.

**exception** circuits.core.manager.**TimeoutError**
Bases: exceptions.Exception

Raised if wait event timeout occurred

**class** circuits.core.manager.**CallValue**(*value*)
Bases: object

**class** circuits.core.manager.**ExceptionWrapper**(*exception*)
Bases: object

**extract**()

**class** circuits.core.manager.**Sleep**(*seconds*)
Bases: *circuits.six.Iterator*

**expired**

**task**

circuits.core.manager.**sleep**(*seconds*)
Delay execution of a coroutine for a given number of seconds. The argument may be a floating point number for subsecond precision.

**class** circuits.core.manager.**Manager**(**args*, ***kwargs*)
Bases: object

The manager class has two roles. As a base class for component implementation, it provides methods for event and handler management. The method *fireEvent()* appends a new event at the end of the event queue for later execution. *waitEvent()* suspends the execution of a handler until all handlers for a given event have been invoked. *callEvent()* combines the last two methods in a single method.

The methods *addHandler()* and *removeHandler()* allow handlers for events to be added and removed dynamically. (The more common way to register a handler is to use the *handler()* decorator or derive the class from *Component*.)

In its second role, the *Manager* takes the role of the event executor. Every component hierarchy has a root component that maintains a queue of events. Firing an event effectively means appending it to the event queue maintained by the root manager. The *flush()* method removes all pending events from the queue and, for each event, invokes all the handlers. Usually, *flush()* is indirectly invoked by *run()*.

The manager optionally provides information about the execution of events as automatically generated events. If an *Event* has its success attribute set to True, the manager fires a Success event if all handlers have

been executed without error. Note that this event will be enqueued (and dispatched) immediately after the events that have been fired by the event's handlers. So the success event indicates both the successful invocation of all handlers for the event and the processing of the immediate follow-up events fired by those handlers.

Sometimes it is not sufficient to know that an event and its immediate follow-up events have been processed. Rather, it is important to know when all state changes triggered by an event, directly or indirectly, have been performed. This also includes the processing of events that have been fired when invoking the handlers for the follow-up events and the processing of events that have again been fired by those handlers and so on. The completion of the processing of an event and all its direct or indirect follow-up events may be indicated by a `Complete` event. This event is generated by the manager if *`Event`* has its `complete` attribute set to True.

Apart from the event queue, the root manager also maintains a list of tasks, actually Python generators, that are updated when the event queue has been flushed.

initializes x; see x.__class__.__doc__ for signature

**name**
Return the name of this Component/Manager

**running**
Return the running state of this Component/Manager

**pid**
Return the process id of this Component/Manager

**getHandlers**(*event*, *channel*, ***kwargs*)

**addHandler**(*f*)

**removeHandler**(*method*, *event=None*)

**registerChild**(*component*)

**unregisterChild**(*component*)

**fireEvent**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**fire**(*event*, **channels*, ***kwargs*)
Fire an event into the system.

**Parameters**

- **event** – The event that is to be fired.
- **channels** – The channels that this event is delivered on. If no channels are specified, the event is delivered to the channels found in the event's `channel` attribute. If this attribute is not set, the event is delivered to the firing component's channel. And eventually, when set neither, the event is delivered on all channels ("*").

**registerTask**(*g*)

**unregisterTask**(*g*)

**waitEvent**(*event*, **channels*, ***kwargs*)

**wait**(*event*, **channels*, ***kwargs*)

**callEvent**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**call**(*event*, **channels*, ***kwargs*)
Fire the given event to the specified channels and suspend execution until it has been dispatched. This method may only be invoked as argument to a `yield` on the top execution level of a handler (e.g. "`yield self.callEvent(event)`"). It effectively creates and returns a generator that will be invoked by the main loop until the event has been dispatched (see *circuits.core.handlers.handler()*).

**flushEvents**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**flush**()
Flush all Events in the Event Queue. If called on a manager that is not the root of an object hierarchy, the invocation is delegated to the root manager.

**start**(*process=False*, *link=None*)
Start a new thread or process that invokes this manager's `run()` method. The invocation of this method returns immediately after the task or process has been started.

**join**()

**stop**(*code=None*)
Stop this manager. Invoking this method causes an invocation of `run()` to return.

**processTask**(*event*, *task*, *parent=None*)

**tick**(*timeout=-1*)
Execute all possible actions once. Process all registered tasks and flush the event queue. If the application is running fire a GenerateEvents to get new events from sources.

This method is usually invoked from *run()*. It may also be used to build an application specific main loop.

**Parameters** **timeout** (*float, measuring seconds*) – the maximum waiting time spent in this method. If negative, the method may block until at least one action has been taken.

**run**(*socket=None*)
Run this manager. The method fires the `Started` event and then continuously calls *tick()*.

The method returns when the manager's *stop()* method is invoked.

If invoked by a programs main thread, a signal handler for the `INT` and `TERM` signals is installed. This handler fires the corresponding `Signal` events and then calls *stop()* for the manager.

### 2.5.13 circuits.core.pollers module

Poller Components for asynchronous file and socket I/O.

This module contains Poller components that enable polling of file or socket descriptors for read/write events. Pollers: - Select - Poll - EPoll

**class** circuits.core.pollers.**BasePoller**(*channel=None*)
Bases: *circuits.core.components.BaseComponent*

**channel = None**

**resume**()

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**isReading**(*fd*)

**isWriting**(*fd*)

**discard**(*fd*)

**getTarget**(*fd*)

**class** circuits.core.pollers.**Select**(...) → new Select Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new Select Poller Component that uses the select poller implementation. This poller is not recommended but is available for legacy reasons as most systems implement select-based polling for backwards compatibility.

**channel = 'select'**

**class** circuits.core.pollers.**Poll**(...) → new Poll Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new Poll Poller Component that uses the poll poller implementation.

**channel = 'poll'**

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**discard**(*fd*)

**class** circuits.core.pollers.**EPoll**(...) → new EPoll Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new EPoll Poller Component that uses the epoll poller implementation.

**channel = 'epoll'**

**addReader**(*source*, *fd*)

**addWriter**(*source*, *fd*)

**removeReader**(*fd*)

**removeWriter**(*fd*)

**discard**(*fd*)

**class** circuits.core.pollers.**KQueue**(...) → new KQueue Poller Component
Bases: *circuits.core.pollers.BasePoller*

Creates a new KQueue Poller Component that uses the kqueue poller implementation.

**channel = 'kqueue'**

**addReader**(*source*, *sock*)

**addWriter**(*source*, *sock*)

**removeReader**(*sock*)

**removeWriter**(*sock*)

**discard**(*sock*)

circuits.core.pollers.**Poller**
alias of *Select*

### 2.5.14 circuits.core.timers module

Timer component to facilitate timed events.

**class** circuits.core.timers.**Timer**(*interval*, *event*, **channels*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Timer Component

A timer is a component that fires an event once after a certain delay or periodically at a regular interval.

**Parameters**

- **interval** (datetime or number of seconds as a float) – the delay or interval to wait for until the event is fired. If interval is specified as datetime, the interval is recalculated as the time span from now to the given datetime.
- **event** (*Event*) – the event to fire.
- **persist** (bool) – An optional keyword argument which if True will cause the event to be fired repeatedly once per configured interval until the timer is unregistered. **Default:** False

**reset**(*interval=None*)
Reset the timer, i.e. clear the amount of time already waited for.

**expiry**

### 2.5.15 circuits.core.utils module

Utils

This module defines utilities used by circuits.

circuits.core.utils.**flatten**(*root*, *visited=None*)

circuits.core.utils.**findchannel**(*root*, *channel*, *all=False*)

circuits.core.utils.**findtype**(*root*, *component*, *all=False*)

circuits.core.utils.**findcmp**(*root*, *component*, *all=False*)

circuits.core.utils.**findroot**(*component*)

circuits.core.utils.**safeimport**(*name*)

### 2.5.16 circuits.core.values module

This defines the Value object used by components and events.

**class** circuits.core.values.**Value**(*event=None*, *manager=None*)
Bases: object

Create a new future Value Object

Creates a new future Value Object which is used by Event Objects and the Manager to store the result(s) of an Event Handler's exeuction of some Event in the system.

**Parameters**

- **event** (*Event instance*) – The Event this Value is associated with.
- **manager** (*A Manager/Component instance.*) – The Manager/Component used to trigger notifications.

**Variables**

- **result** – True if this value has been changed.
- ***errors*** – True if while setting this value an exception occured.
- ***notify*** – True or an event name to notify of changes to this value

This is a Future/Promise implementation.

**inform**(*force=False*)

**getValue**(*recursive=True*)

**setValue**(*value*)

**value**
Value of this Value

### 2.5.17 circuits.core.workers module

Workers

Worker components used to perform "work" in independent threads or processes. Worker(s) are typically used by a Pool (circuits.core.pools) to create a pool of workers. Worker(s) are not registered with a Manager or another Component - instead they are managed by the Pool. If a Worker is used independently it should not be registered as it causes its main event handler _on_task to execute in the other thread blocking it.

**class** circuits.core.workers.**task**(*f*, **args*, ***kwargs*)
Bases: *circuits.core.events.Event*

task Event

This Event is used to initiate a new task to be performed by a Worker or a Pool of Worker(s).

**Parameters**

- **f** (*function*) – The function to be executed.
- **args** (*tuple*) – Arguments to pass to the function
- **kwargs** (*dict*) – Keyword Arguments to pass to the function

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**success = True**

**failure** = True

**name** = 'task'

**class** circuits.core.workers.**Worker**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

A thread/process Worker Component

This Component creates a Worker (either a thread or process) which when given a Task, will execute the given function in the task in the background in its thread/process.

**Parameters** **process** (*bool*) – True to start this Worker as a process (Thread otherwise)

initializes x; see x.__class__.__doc__ for signature

**channel** = 'worker'

**init**(*process=False*, *workers=None*, *channel='worker'*)

## 2.5.18 circuits.io package

### Submodules

#### circuits.io.events module

I/O Events

This module implements commonly used I/O events used by other I/O modules.

**class** circuits.io.events.**accessed**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

accessed Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to True, an associated event success (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'accessed'**

**class** `circuits.io.events.`**close**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

close Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'close'**

**class** `circuits.io.events.`**closed**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

closed Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'closed'**

**class** `circuits.io.events.`**created**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

created Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'created'**

**class** `circuits.io.events.`**deleted**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

deleted Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'deleted'**

**class** circuits.io.events.**eof**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

eof Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'eof'**

**class** circuits.io.events.**error**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

error Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to

as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'error'**

**class** `circuits.io.events.`**modified**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

modified Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'modified'**

**class** `circuits.io.events.`**moved**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

moved Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'moved'**

**class** `circuits.io.events.`**open**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

open Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'open'**

**class** `circuits.io.events.`**opened**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

opened Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'opened'**

**class** `circuits.io.events.`**read**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

read Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'read'**

**class** circuits.io.events.**ready**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

ready Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'ready'**

**class** circuits.io.events.**seek**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

seek Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'seek'**

**class** `circuits.io.events.`**started**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

started Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'started'**

**class** circuits.io.events.**stopped**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

stopped Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'stopped'**

**class** circuits.io.events.**unmounted**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

unmounted Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'unmounted'**

**class** `circuits.io.events.`**write**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

write Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'write'**

### circuits.io.file module

File I/O

This module implements a wrapper for basic File I/O.

**class** circuits.io.file.**File**(**args*, ***kwargs*)

Bases: *circuits.core.components.Component*

initializes x; see x.__class__.__doc__ for signature

**channel = 'file'**

**close**()

**closed**

**filename**

**init**(*filename*, *mode='r'*, *bufsize=4096*, *encoding=None*, *channel='file'*)

**mode**

**seek**(*offset*, *whence=0*)

**write**(*data*)

### circuits.io.notify module

### circuits.io.process module

Process

This module implements a wrapper for basic `subprocess.Popen` functionality.

**class** circuits.io.process.**Process**(**args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'process'**

**init**(*args*, *cwd=None*, *shell=False*)

**kill**()

**signal**(*signal*)

**start**()

**status**

**stop**()

**wait**()

**write**(*data*)

#### circuits.io.serial module

Serial I/O

This module implements basic Serial (RS232) I/O.

**class** circuits.io.serial.**Serial**(*port*, *baudrate=115200*, *bufsize=4096*, *timeout=0.2*, *channel='serial'*)

Bases: *circuits.core.components.Component*

**channel = 'serial'**

**close**()

**write**(*data*)

### Module contents

I/O Support

This package contains various I/O Components. Provided are a generic File Component, StdIn, StdOut and StdErr components. Instances of StdIn, StdOut and StdErr are also created by importing this package.

## 2.5.19 circuits.io.events module

I/O Events

This module implements commonly used I/O events used by other I/O modules.

**class** circuits.io.events.**accessed**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

accessed Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'accessed'**

**class** `circuits.io.events.`**close**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

close Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'close'**

**class** circuits.io.events.**closed**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

closed Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'closed'**

**class** circuits.io.events.**created**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

created Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to

as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'created'**

**class** `circuits.io.events.`**deleted**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

deleted Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'deleted'**

**class** `circuits.io.events.`**eof**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

eof Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'eof'**

**class** `circuits.io.events.`**error**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

error Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'error'**

**class** `circuits.io.events.`**`modified`**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

modified Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'modified'**

**class** `circuits.io.events.`**moved**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

moved Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'moved'**

**class** circuits.io.events.**open**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

open Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'open'**

**class** circuits.io.events.**opened**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

opened Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'opened'**

**class** `circuits.io.events.`**read**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

read Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'read'**

**class** circuits.io.events.**ready**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

ready Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'ready'**

**class** circuits.io.events.**seek**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

seek Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'seek'**

**class** `circuits.io.events.`**started**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

started Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'started'**

**class** `circuits.io.events.`**stopped**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

stopped Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'stopped'**

**class** `circuits.io.events.`**unmounted**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

unmounted Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'unmounted'**

**class** `circuits.io.events.`**write**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

write Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'write'**

### 2.5.20 circuits.io.file module

File I/O

This module implements a wrapper for basic File I/O.

**class** `circuits.io.file.`**File**(**args*, ***kwargs*)

Bases: *circuits.core.components.Component*

initializes x; see x.__class__.__doc__ for signature

**channel = 'file'**

**close**()

**closed**

**filename**

**init**(*filename*, *mode='r'*, *bufsize=4096*, *encoding=None*, *channel='file'*)

**mode**

**seek**(*offset*, *whence=0*)

**write**(*data*)

### 2.5.21 circuits.io.notify module

### 2.5.22 circuits.io.process module

Process

This module implements a wrapper for basic `subprocess.Popen` functionality.

**class** `circuits.io.process.`**Process**(**args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'process'**

**init**(*args*, *cwd=None*, *shell=False*)

**kill**()

**signal**(*signal*)

**start**()

**status**

**stop**()

**wait**()

**write**(*data*)

### 2.5.23 circuits.io.serial module

Serial I/O

This module implements basic Serial (RS232) I/O.

**class** circuits.io.serial.**Serial**(*port*, *baudrate=115200*, *bufsize=4096*, *timeout=0.2*, *channel='serial'*)

Bases: *circuits.core.components.Component*

**channel = 'serial'**

**close**()

**write**(*data*)

### 2.5.24 circuits.net package

#### Submodules

##### circuits.net.events module

Networking Events

This module implements commonly used Networking events used by socket components.

**class** circuits.net.events.**broadcast**(**args*)

Bases: *circuits.core.events.Event*

broadcast Event

This Event is used by the UDPServer/UDPClient sockets to send a message on the <broadcast> network.

---

**Note:**

- This event is never sent, it is used to send data.
- This event is used for both Client and Server UDP Components.

---

**Parameters** **args** – (data, port)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'broadcast'**

**class** circuits.net.events.**close**(**args*)
Bases: *circuits.core.events.Event*

close Event

This Event is used to notify a client, client connection or server that we want to close.

---

**Note:**

•This event is never sent, it is used to close.

•This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: () Server: (sock)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'close'**

**class** circuits.net.events.**closed**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

closed Event

This Event is sent when a server has closed its listening socket.

---

**Note:** This event is for Server components.

---

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'closed'**

**class** circuits.net.events.**connect**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

connect Event

This Event is sent when a new client connection has arrived on a server. This event is also used for client's to initiate a new connection to a remote host.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters**

- **args** (*tuple*) – Client: (host, port) Server: (sock, host, port)
- **kwargs** (*dict*) – Client: (ssl)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'connect'**

**class** circuits.net.events.**connected**(*host*, *port*)
Bases: *circuits.core.events.Event*

connected Event

This Event is sent when a client has successfully connected.

---

**Note:** This event is for Client Components.

---

**Parameters**

- **host** – The hostname connected to.
- **port** – The port connected to

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'connected'**

**class** circuits.net.events.**disconnect**(**args*)
Bases: *circuits.core.events.Event*

disconnect Event

This Event is sent when a client connection has closed on a server. This event is also used for client's to disconnect from a remote host.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: () Server: (sock)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'disconnect'**

**class** circuits.net.events.**disconnected**
Bases: *circuits.core.events.Event*

disconnected Event

This Event is sent when a client has disconnected

---

**Note:** This event is for Client Components.

---

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'disconnected'**

**class** circuits.net.events.**error**(**args*)
Bases: *circuits.core.events.Event*

error Event

This Event is sent when a client or server connection has an error.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (error) Server: (sock, error)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'error'**

**class** circuits.net.events.**read**(**args*)
Bases: *circuits.core.events.Event*

read Event

This Event is sent when a client or server connection has read any data.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (data) Server: (sock, data)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'read'**

**class** circuits.net.events.**ready**(*component*, *bind=None*)
Bases: *circuits.core.events.Event*

ready Event

This Event is used to notify the rest of the system that the underlying Client or Server Component is ready to begin processing connections or incoming/outgoing data. (This is triggered as a direct result of having the capability to support multiple client/server components with a single poller component instance in a system).

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters**

- **component** – The Client/Server Component that is ready.
- **bind** – The (host, port) the server has bound to.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'ready'**

**class** circuits.net.events.**unreachable**(*host*, *port*, *reason=None*)
Bases: *circuits.core.events.Event*

unreachable Event

This Event is sent when a server is unreachable for a client

**Parameters**

- **host** – Server hostname or IP
- **port** – Server port

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'unreachable'**

**class** circuits.net.events.**write**(**args*)
Bases: *circuits.core.events.Event*

write Event

This Event is used to notify a client, client connection or server that we have data to be written.

---

**Note:**

•This event is never sent, it is used to send data.

•This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (data) Server: (sock, data)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'write'**

### circuits.net.sockets module

Socket Components

This module contains various Socket Components for use with Networking.

**class** circuits.net.sockets.**Client**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'client'**

**close**()

**connected**

**parse_bind_parameter**(*bind_parameter*)

**write**(*data*)

circuits.net.sockets.**Pipe**(**channels*, ***kwargs*)
Create a new full duplex Pipe

Returns a pair of UNIXClient instances connected on either side of the pipe.

**class** circuits.net.sockets.**Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'server'**

**close**(*sock=None*)

**connected**

**host**

**parse_bind_parameter**(*bind_parameter*)

**port**

**write**(*sock*, *data*)

**class** circuits.net.sockets.**TCP6Client**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.TCPClient*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

**class** circuits.net.sockets.**TCP6Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.TCPServer*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

**class** circuits.net.sockets.**TCPClient**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.Client*

**connect**(*host*, *port*, *secure=False*, ***kwargs*)

**init**(*connect_timeout=5*, **args*, ***kwargs*)

**socket_family = 2**

**class** circuits.net.sockets.**TCPServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 2**

circuits.net.sockets.**UDP6Client**
alias of *UDP6Server*

**class** circuits.net.sockets.**UDP6Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.UDPServer*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

circuits.net.sockets.**UDPClient**
alias of *UDPServer*

**class** circuits.net.sockets.**UDPServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

**broadcast** (*data*, *port*)

**close** ()

**socket_family** = 2

**write** (*address*, *data*)

**class** circuits.net.sockets.**UNIXClient** (*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.Client*

**connect** (*path*, *secure=False*, ***kwargs*)

**ready** (*component*)

**class** circuits.net.sockets.**UNIXServer** (*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

circuits.net.sockets.**do_handshake** (*sock*, *on_done=None*, *on_error=None*, *extra_args=None*)
SSL Async Handshake

**Parameters**

- **on_done** – Function called when handshake is complete
- **on_error** – Function called when handshake errored

circuits.net.sockets.**parse_ipv4_parameter** (*bind_parameter*)

circuits.net.sockets.**parse_ipv6_parameter** (*bind_parameter*)

### Module contents

Networking Components

This package contains components that implement network sockets and protocols for implementing client and server network applications.

## 2.5.25 circuits.net.events module

Networking Events

This module implements commonly used Networking events used by socket components.

**class** circuits.net.events.**broadcast** (**args*)
Bases: *circuits.core.events.Event*

broadcast Event

This Event is used by the UDPServer/UDPClient sockets to send a message on the <broadcast> network.

---

**Note:**

•This event is never sent, it is used to send data.

•This event is used for both Client and Server UDP Components.

---

**Parameters** **args** – (data, port)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'broadcast'**

**class** circuits.net.events.**close**(**args*)
Bases: *circuits.core.events.Event*

close Event

This Event is used to notify a client, client connection or server that we want to close.

---

**Note:**

•This event is never sent, it is used to close.

•This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: () Server: (sock)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'close'**

**class** circuits.net.events.**closed**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

closed Event

This Event is sent when a server has closed its listening socket.

---

**Note:** This event is for Server components.

---

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'closed'**

**class** circuits.net.events.**connect**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

connect Event

This Event is sent when a new client connection has arrived on a server. This event is also used for client's to initiate a new connection to a remote host.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters**

- **args** (*tuple*) – Client: (host, port) Server: (sock, host, port)
- **kwargs** (*dict*) – Client: (ssl)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'connect'**

**class** circuits.net.events.**connected**(*host*, *port*)
Bases: *circuits.core.events.Event*

connected Event

This Event is sent when a client has successfully connected.

---

**Note:** This event is for Client Components.

---

**Parameters**

- **host** – The hostname connected to.
- **port** – The port connected to

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'connected'**

**class** circuits.net.events.**disconnect**(**args*)
Bases: *circuits.core.events.Event*

disconnect Event

This Event is sent when a client connection has closed on a server. This event is also used for client's to disconnect from a remote host.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: () Server: (sock)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'disconnect'**

**class** circuits.net.events.**disconnected**
Bases: *circuits.core.events.Event*

disconnected Event

This Event is sent when a client has disconnected

---

**Note:** This event is for Client Components.

---

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'disconnected'**

**class** circuits.net.events.**error**(**args*)
Bases: *circuits.core.events.Event*

error Event

This Event is sent when a client or server connection has an error.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (error) Server: (sock, error)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'error'**

**class** circuits.net.events.**read**(**args*)
Bases: *circuits.core.events.Event*

read Event

This Event is sent when a client or server connection has read any data.

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (data) Server: (sock, data)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'read'**

**class** circuits.net.events.**ready**(*component*, *bind=None*)
Bases: *circuits.core.events.Event*

ready Event

This Event is used to notify the rest of the system that the underlying Client or Server Component is ready to begin processing connections or incoming/outgoing data. (This is triggered as a direct result of having the capability to support multiple client/server components with a single poller component instance in a system).

---

**Note:** This event is used for both Client and Server Components.

---

**Parameters**

- **component** – The Client/Server Component that is ready.
- **bind** – The (host, port) the server has bound to.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'ready'**

**class** circuits.net.events.**unreachable**(*host*, *port*, *reason=None*)
Bases: *circuits.core.events.Event*

unreachable Event

This Event is sent when a server is unreachable for a client

**Parameters**

- **host** – Server hostname or IP
- **port** – Server port

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'unreachable'**

**class** circuits.net.events.**write**(**args*)
Bases: *circuits.core.events.Event*

write Event

This Event is used to notify a client, client connection or server that we have data to be written.

---

**Note:**

•This event is never sent, it is used to send data.

•This event is used for both Client and Server Components.

---

**Parameters** **args** – Client: (data) Server: (sock, data)

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'write'**

### 2.5.26 circuits.net.sockets module

Socket Components

This module contains various Socket Components for use with Networking.

**class** circuits.net.sockets.**Client**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'client'**

**close**()

**connected**

**parse_bind_parameter**(*bind_parameter*)

**write**(*data*)

circuits.net.sockets.**Pipe**(**channels*, ***kwargs*)
Create a new full duplex Pipe

Returns a pair of UNIXClient instances connected on either side of the pipe.

**class** circuits.net.sockets.**Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'server'**

**close**(*sock=None*)

**connected**

**host**

**parse_bind_parameter**(*bind_parameter*)

**port**

**write**(*sock*, *data*)

**class** circuits.net.sockets.**TCP6Client**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.TCPClient*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

**class** circuits.net.sockets.**TCP6Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.TCPServer*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

**class** circuits.net.sockets.**TCPClient**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.Client*

**connect**(*host*, *port*, *secure=False*, ***kwargs*)

**init**(*connect_timeout=5*, **args*, ***kwargs*)

**socket_family = 2**

**class** circuits.net.sockets.**TCPServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 2**

circuits.net.sockets.**UDP6Client**
alias of *UDP6Server*

**class** circuits.net.sockets.**UDP6Server**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.UDPServer*

**parse_bind_parameter**(*bind_parameter*)

**socket_family = 10**

circuits.net.sockets.**UDPClient**
alias of *UDPServer*

**class** circuits.net.sockets.**UDPServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

**broadcast**(*data*, *port*)

**close**()

**socket_family** = 2

**write**(*address*, *data*)

**class** circuits.net.sockets.**UNIXClient**(*bind=None*, *bufsize=4096*, *channel='client'*, ***kwargs*)
Bases: *circuits.net.sockets.Client*

**connect**(*path*, *secure=False*, ***kwargs*)

**ready**(*component*)

**class** circuits.net.sockets.**UNIXServer**(*bind*, *secure=False*, *backlog=5000*, *bufsize=4096*, *channel='server'*, ***kwargs*)
Bases: *circuits.net.sockets.Server*

circuits.net.sockets.**do_handshake**(*sock*, *on_done=None*, *on_error=None*, *extra_args=None*)
SSL Async Handshake

**Parameters**

- **on_done** – Function called when handshake is complete
- **on_error** – Function called when handshake errored

circuits.net.sockets.**parse_ipv4_parameter**(*bind_parameter*)

circuits.net.sockets.**parse_ipv6_parameter**(*bind_parameter*)

## 2.5.27 circuits.node package

### Submodules

#### circuits.node.client module

**class** circuits.node.client.**Client**(*host*, *port*, *channel='node_client'*, *receive_event_firewall=None*, *send_event_firewall=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Node Client (peer)

Create new connection for a node.

**Parameters**

- **hostname** (*str*) – hostname to connect.
- **port** (*int*) – port to connect.
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** `node_client`
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for sending. **Default:** `None` (no firewall)

- **send_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for executing **Default:** `None` (no firewall)

**channel = 'node_client'**

**close**()
Close the connection

**connect**()
Create the connection

**send**(*event*)
Send event through the connection

**Parameters** **event** (*circuits.core.events.Event*) – Event to send.

**Returns** The result object of the sended event

**Return type** generator

### circuits.node.events module

**class** circuits.node.events.**connected_to**(*connection_name*, *hostname*, *port*, *client_channel*, *client_obj*)
Bases: *circuits.core.events.Event*

Connected to a peer

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote system.
- **port** (*int*) – connection port of the remote system.
- **client_channel** (*str*) – Channel used for client event.
- **client_obj** (*circuits.net.sockets.Client*) – Client object.

**name = 'connected_to'**

**class** circuits.node.events.**disconnected_from**(*connection_name*, *hostname*, *port*, *client_channel*, *client_obj*)
Bases: *circuits.node.events.connected_to*

Disconnected from a peer

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote system.
- **port** (*int*) – connection port of the remote system.
- **client_channel** (*str*) – Channel used for client event.
- **client_obj** (*circuits.net.sockets.Client*) – Client object.

**name = 'disconnected_from'**

**class** circuits.node.events.**remote**(*event*, *connection_name*, *channel=None*)
Bases: *circuits.core.events.Event*

send event to a peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **connection_name** (*str*) – Connection name.
- **channel** (*str*) – Remote channel (channel to use on peer).

**name = 'remote'**

### circuits.node.node module

Node

this module manage node (start server, add peer, ...) .. seealso:: Examples in **:dir:'examples.node'**

**class** circuits.node.node.**Node**(*port=None*, *channel='node'*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

this class manage node (start server, add peer, ...) .. seealso:: Examples in **:dir:'examples.node'**

Start node system.

**Parameters**

- **port** (*int*) – An optional keyword argument which if defined, start server on this port. **Default:** `None` (don't start the server)
- **server_ip** (*str*) – An optional keyword argument which define ip where the socket has listen to. **Default:** `0.0.0.0` (all ip is allowed)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** `node`
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for sending. **Default:** `None` (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for executing **Default:** `None` (no firewall)

**add**(*connection_name*, *hostname*, *port*, ***kwargs*)
Add new peer to the node.

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote node.
- **port** (*int*) – port of the remote node.
- **auto_remote_event** (*dict*) – An optional keyword argument which if defined, bind events automatically to remote execution. **Default:** `{}` (no events)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for client event. If this keyword is not defined the method will generate the channel name automatically.

- **reconnect_delay** (*int*) – An optional keyword argument which if defined, set auto reconnect delay. **Default:** `10` (seconde)
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for sending. **Default:** `None` (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, setfunction or method to call to check if event is allowed for executing **Default:** `None` (no firewall)

**Returns** Channel used on client event.

**Return type** str

**channel = 'node'**

**get_connection_names**()
Get connections names

**Returns** The list of connections names

**Return type** list of str

**get_peer**(*connection_name*)
Get a client object by name

**Parameters** **connection_name** (*str*) – Connection name.

**Returns** The Client object

**Return type** *circuits.node.client.Client*

### circuits.node.server module

**class** circuits.node.server.**Server**(*port*, *server_ip='0.0.0.0'*, *channel='node'*, *receive_event_firewall=None*, *send_event_firewall=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Node server.

Create server on node system.

**Parameters**

- **port** (*int*) – start server on this port.
- **server_ip** (*str*) – An optional keyword argument which which define ip where the socket has listen to. **Default:** `0.0.0.0` (all ip is allowed)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** `node`
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for sending **Default:** `None` (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for executing **Default:** `None` (no firewall)

**channel = 'node'**

**get_socks**()
Get clients sockets list

**Returns** The list of client socket

**Return type** list of `socket.socket`

**host**

**port**

**send**(*event*, *sock*, *no_result=False*)
Send event to peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **sock** (`socket.socket`) – Client's socket (peer selection).
- **no_result** (*bool*) – An optional keyword argument which if True don't return the event result. **Default:** `False` (wait the result)

**Returns** The result of remote event

**Return type** generator

**send_all**(*event*)
Send event to all peer

**Parameters** **event** (*circuits.core.events.Event*) – Event to execute remotely.

**send_to**(*event*, *socks*)
Send event to multiple peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **socks** (list of `socket.socket`) – Client's socket list (peer selection).

### circuits.node.utils module

circuits.node.utils.**dump_event**(*e*, *id*)

circuits.node.utils.**dump_value**(*v*)

circuits.node.utils.**load_event**(*s*)

circuits.node.utils.**load_value**(*v*)

## Module contents

Node

Distributed processing support for circuits.

### 2.5.28 circuits.node.client module

**class** circuits.node.client.**Client**(*host*, *port*, *channel='node_client'*, *receive_event_firewall=None*, *send_event_firewall=None*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

Node Client (peer)

Create new connection for a node.

**Parameters**

- **hostname** (*str*) – hostname to connect.
- **port** (*int*) – port to connect.
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** `node_client`
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for sending. **Default:** `None` (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for executing **Default:** `None` (no firewall)

**channel = 'node_client'**

**close**()
Close the connection

**connect**()
Create the connection

**send**(*event*)
Send event through the connection

**Parameters** **event** (*circuits.core.events.Event*) – Event to send.

**Returns** The result object of the sended event

**Return type** generator

### 2.5.29 circuits.node.events module

**class** circuits.node.events.**connected_to**(*connection_name*, *hostname*, *port*, *client_channel*, *client_obj*)

Bases: *circuits.core.events.Event*

Connected to a peer

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote system.
- **port** (*int*) – connection port of the remote system.
- **client_channel** (*str*) – Channel used for client event.
- **client_obj** (*circuits.net.sockets.Client*) – Client object.

**name = 'connected_to'**

**class** circuits.node.events.**disconnected_from**(*connection_name*, *hostname*, *port*, *client_channel*, *client_obj*)

Bases: *circuits.node.events.connected_to*

Disconnected from a peer

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote system.
- **port** (*int*) – connection port of the remote system.
- **client_channel** (*str*) – Channel used for client event.
- **client_obj** (*circuits.net.sockets.Client*) – Client object.

**name = 'disconnected_from'**

**class** circuits.node.events.**remote**(*event*, *connection_name*, *channel=None*)

Bases: *circuits.core.events.Event*

send event to a peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **connection_name** (*str*) – Connection name.
- **channel** (*str*) – Remote channel (channel to use on peer).

**name = 'remote'**

### 2.5.30 circuits.node.node module

Node

this module manage node (start server, add peer, ...) .. seealso:: Examples in **:dir:'examples.node'**

**class** circuits.node.node.**Node**(*port=None*, *channel='node'*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

this class manage node (start server, add peer, ...) .. seealso:: Examples in **:dir:'examples.node'**

Start node system.

**Parameters**

- **port** (*int*) – An optional keyword argument which if defined, start server on this port. **Default:** `None` (don't start the server)
- **server_ip** (*str*) – An optional keyword argument which define ip where the socket has listen to. **Default:** `0.0.0.0` (all ip is allowed)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** `node`
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for sending. **Default:** `None` (no firewall)

- **send_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for executing **Default:** `None` (no firewall)

**add**(*connection_name*, *hostname*, *port*, ***kwargs*)
Add new peer to the node.

**Parameters**

- **connection_name** (*str*) – Connection name.
- **hostname** (*str*) – hostname of the remote node.
- **port** (*int*) – port of the remote node.
- **auto_remote_event** (*dict*) – An optional keyword argument which if defined, bind events automatically to remote execution. **Default:** `{}` (no events)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for client event. If this keyword is not defined the method will generate the channel name automatically.
- **reconnect_delay** (*int*) – An optional keyword argument which if defined, set auto reconnect delay. **Default:** `10` (seconde)
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, function or method to call for check if event is allowed for sending. **Default:** `None` (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, setfunction or method to call to check if event is allowed for executing **Default:** `None` (no firewall)

**Returns** Channel used on client event.

**Return type** str

**channel = 'node'**

**get_connection_names**()
Get connections names

**Returns** The list of connections names

**Return type** list of str

**get_peer**(*connection_name*)
Get a client object by name

**Parameters** **connection_name** (*str*) – Connection name.

**Returns** The Client object

**Return type** *circuits.node.client.Client*

### 2.5.31 circuits.node.server module

**class** circuits.node.server.**Server**(*port*, *server_ip='0.0.0.0'*, *channel='node'*, *receive_event_firewall=None*, *send_event_firewall=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Node server.

Create server on node system.

**Parameters**

- **port** (*int*) – start server on this port.
- **server_ip** (*str*) – An optional keyword argument which which define ip where the socket has listen to. **Default:** `0.0.0.0` (all ip is allowed)
- **channel** (*str*) – An optional keyword argument which if defined, set channel used for node event. **Default:** `node`
- **receive_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for sending **Default:** `None` (no firewall)
- **send_event_firewall** (*method*) – An optional keyword argument which if defined, set function or method to call to check if event is allowed for executing **Default:** `None` (no firewall)

**channel = 'node'**

**get_socks**()
Get clients sockets list

**Returns** The list of client socket

**Return type** list of `socket.socket`

**host**

**port**

**send**(*event*, *sock*, *no_result=False*)
Send event to peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **sock** (`socket.socket`) – Client's socket (peer selection).
- **no_result** (*bool*) – An optional keyword argument which if True don't return the event result. **Default:** `False` (wait the result)

**Returns** The result of remote event

**Return type** generator

**send_all**(*event*)
Send event to all peer

**Parameters** **event** (*circuits.core.events.Event*) – Event to execute remotely.

**send_to**(*event*, *socks*)
Send event to multiple peer

**Parameters**

- **event** (*circuits.core.events.Event*) – Event to execute remotely.
- **socks** (list of `socket.socket`) – Client's socket list (peer selection).

### 2.5.32 circuits.node.utils module

circuits.node.utils.**dump_event**(*e*, *id*)

circuits.node.utils.**dump_value**(*v*)

circuits.node.utils.**load_event**(*s*)

circuits.node.utils.**load_value**(*v*)

### 2.5.33 circuits.protocols package

#### Submodules

##### circuits.protocols.http module

**class** circuits.protocols.http.**HTTP**(*encoding='utf-8'*, *channel='web'*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.protocols.http.**ResponseObject**(*headers*, *status*, *version*)
Bases: object

**read**()

**class** circuits.protocols.http.**request**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

request Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to True, an associated event success (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'request'**

**class** `circuits.protocols.http.`**response**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

response Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'response'**

### circuits.protocols.irc module

Internet Relay Chat Protocol

This package implements the Internet Relay Chat Protocol or commonly known as IRC. Support for both server and client is implemented.

### circuits.protocols.line module

Line Protocol

This module implements the basic Line protocol.

This module can be used in both server and client implementations.

**class** circuits.protocols.line.**Line**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

Line Protocol

Implements the Line Protocol.

Incoming data is split into lines with a splitter function. For each line of data processed a Line Event is created. Any unfinished lines are appended into an internal buffer.

A custom line splitter function can be passed to customize how data is split into lines. This function must accept two arguments, the data to process and any left over data from a previous invocation of the splitter function. The function must also return a tuple of two items, a list of lines and any left over data.

**Parameters** **splitter** (*function*) – a line splitter function

This Component operates in two modes. In normal operation it's expected to be used in conjunction with components that expose a Read Event on a "read" channel with only one argument (data). Some builtin components that expose such events are: - circuits.net.sockets.TCPClient - circuits.io.File

The second mode of operation works with circuits.net.sockets.Server components such as TCPServer, UNIXServer, etc. It's expected that two arguments exist in the Read Event, sock and data. The following two arguments can be passed to affect how unfinished data is stored and retrieved for such components:

**Parameters** **getBuffer** (*function*) – function to retrieve the buffer for a client sock

This function must accept one argument (sock,) the client socket whoose buffer is to be retrieved.

**Parameters** **updateBuffer** (*function*) – function to update the buffer for a client sock

This function must accept two arguments (sock, buffer,) the client socket and the left over buffer to be updated.

**@note: This Component must be used in conjunction with a Component that** exposes Read events on a "read" Channel.

initializes x; see x.__class__.__doc__ for signature

**class** circuits.protocols.line.**line**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

line Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'line'**

`circuits.protocols.line.`**splitLines**(*s*, *buffer*) → lines, buffer

Append s to buffer and find any new lines of text in the string splitting at the standard IRC delimiter CRLF. Any new lines found, return them as a list and the remaining buffer for further processing.

### circuits.protocols.websocket module

**class** `circuits.protocols.websocket.`**WebSocketCodec**(*sock=None*, *data=bytearray(b'')*, **args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

WebSocket Protocol

Implements the Data Framing protocol for WebSocket.

This component is used in conjunction with a parent component that receives Read events on its channel. When created (after a successful WebSocket setup handshake), the codec registers a handler on the parent's channel that filters out these Read events for a given socket (if used in a server) or all Read events (if used in a client). The data is decoded and the contained payload is emitted as Read events on the codec's channel.

The data from write events sent to the codec's channel (with socket argument if used in a server) is encoded according to the WebSocket Data Framing protocol. The encoded data is then forwarded as write events on the parents channel.

Creates a new codec.

**Parameters** **sock** – the socket used in Read and write events (if used in a server, else None)

**channel = 'ws'**

## Module contents

Networking Protocols

This package contains components that implement various networking protocols.

### 2.5.34 circuits.protocols.http module

**class** circuits.protocols.http.**HTTP**(*encoding='utf-8'*, *channel='web'*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.protocols.http.**ResponseObject**(*headers*, *status*, *version*)
Bases: object

**read**()

**class** circuits.protocols.http.**request**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

request Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to True, an associated event success (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to True, an associated event complete (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'request'**

**class** circuits.protocols.http.**response**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

response Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'response'**

### 2.5.35 circuits.protocols.irc module

Internet Relay Chat Protocol

This package implements the Internet Relay Chat Protocol or commonly known as IRC. Support for both server and client is implemented.

### 2.5.36 circuits.protocols.line module

Line Protocol

This module implements the basic Line protocol.

This module can be used in both server and client implementations.

**class** `circuits.protocols.line.`**Line**(**args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

Line Protocol

Implements the Line Protocol.

Incoming data is split into lines with a splitter function. For each line of data processed a Line Event is created. Any unfinished lines are appended into an internal buffer.

A custom line splitter function can be passed to customize how data is split into lines. This function must accept two arguments, the data to process and any left over data from a previous invocation of the splitter function. The function must also return a tuple of two items, a list of lines and any left over data.

**Parameters** **splitter** (*function*) – a line splitter function

This Component operates in two modes. In normal operation it's expected to be used in conjunction with components that expose a Read Event on a "read" channel with only one argument (data). Some builtin components that expose such events are: - circuits.net.sockets.TCPClient - circuits.io.File

The second mode of operation works with circuits.net.sockets.Server components such as TCPServer, UNIXServer, etc. It's expected that two arguments exist in the Read Event, sock and data. The following two arguments can be passed to affect how unfinished data is stored and retrieved for such components:

**Parameters** **getBuffer** (*function*) – function to retrieve the buffer for a client sock

This function must accept one argument (sock,) the client socket whoose buffer is to be retrieved.

**Parameters** **updateBuffer** (*function*) – function to update the buffer for a client sock

This function must accept two arguments (sock, buffer,) the client socket and the left over buffer to be updated.

**@note: This Component must be used in conjunction with a Component that** exposes Read events on a "read" Channel.

initializes x; see x.__class__.__doc__ for signature

**class** circuits.protocols.line.**line**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

line Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'line'**

circuits.protocols.line.**splitLines**(*s*, *buffer*) → lines, buffer

Append s to buffer and find any new lines of text in the string splitting at the standard IRC delimiter CRLF. Any new lines found, return them as a list and the remaining buffer for further processing.

### 2.5.37 circuits.protocols.websocket module

**class** circuits.protocols.websocket.**WebSocketCodec**(*sock=None*, *data=bytearray(b'')*, **args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

WebSocket Protocol

Implements the Data Framing protocol for WebSocket.

This component is used in conjunction with a parent component that receives Read events on its channel. When created (after a successful WebSocket setup handshake), the codec registers a handler on the parent's channel that filters out these Read events for a given socket (if used in a server) or all Read events (if used in a client). The data is decoded and the contained payload is emitted as Read events on the codec's channel.

The data from write events sent to the codec's channel (with socket argument if used in a server) is encoded according to the WebSocket Data Framing protocol. The encoded data is then forwarded as write events on the parents channel.

Creates a new codec.

**Parameters** **sock** – the socket used in Read and write events (if used in a server, else None)

**channel = 'ws'**

### 2.5.38 circuits.six module

Utilities for writing code that runs on Python 2 and 3

circuits.six.**byteindex**(*data*, *index*)

circuits.six.**iterbytes**(*data*)

**class** circuits.six.**MovedModule**(*name*, *old*, *new=None*)

Bases: circuits.six._LazyDescr

**class** circuits.six.**MovedAttribute**(*name*, *old_mod*, *new_mod*, *old_attr=None*, *new_attr=None*)

Bases: circuits.six._LazyDescr

circuits.six.**add_move**(*move*)

Add an item to six.moves.

circuits.six.**remove_move**(*name*)

Remove item from six.moves.

circuits.six.**create_bound_method**(*function*, *instance*)

circuits.six.**get_unbound_function**(*unbound*)
Get the function out of a possibly unbound function

**class** circuits.six.**Iterator**
Bases: object

**next**()

circuits.six.**iterkeys**(*d*)
Return an iterator over the keys of a dictionary.

circuits.six.**itervalues**(*d*)
Return an iterator over the values of a dictionary.

circuits.six.**iteritems**(*d*)
Return an iterator over the (key, value) pairs of a dictionary.

circuits.six.**b**(*s*, *encoding='utf-8'*)
Byte literal

circuits.six.**u**(*s*, *encoding='utf-8'*)
Text literal

circuits.six.**bytes_to_str**(*s*)

circuits.six.**reraise**(*tp*, *value*, *tb=None*)
Reraise an exception.

circuits.six.**exec_**(*code*, *globs=None*, *locs=None*)
Execute code in a namespace.

circuits.six.**print_**(**args*, ***kwargs*)
The new-style print function.

circuits.six.**with_metaclass**(*meta*, *base=<type 'object'>*)
Create a base class with a metaclass.

### 2.5.39 circuits.tools package

#### Module contents

Circuits Tools

circuits.tools contains a standard set of tools for circuits. These tools are installed as executables with a prefix of "circuits."

circuits.tools.**tryimport**(*modules*, *obj=None*, *message=None*)

circuits.tools.**walk**(*x*, *f*, *d=0*, *v=None*)

circuits.tools.**edges**(*x*, *e=None*, *v=None*, *d=0*)

circuits.tools.**findroot**(*x*)

circuits.tools.**kill**(*x*)

circuits.tools.**graph**(*x*, *name=None*)
Display a directed graph of the Component structure of x

**Parameters**

- **x** (*Component or Manager*) – A Component or Manager to graph
- **name** (*str*) – A name for the graph (defaults to x's name)

@return: A directed graph representing x's Component structure. @rtype: str

circuits.tools.**inspect**(*x*)

Display an inspection report of the Component or Manager x

**Parameters** **x** (*Component or Manager*) – A Component or Manager to graph

@return: A detailed inspection report of x @rtype: str

circuits.tools.**deprecated**(*f*)

## 2.5.40 circuits.version module

## 2.5.41 circuits.web package

### Subpackages

#### circuits.web.dispatchers package

##### Submodules

**circuits.web.dispatchers.dispatcher module** Dispatcher

This module implements a basic URL to Channel dispatcher. This is the default dispatcher used by circuits.web

**class** circuits.web.dispatchers.dispatcher.**Dispatcher**(***kwargs*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

circuits.web.dispatchers.dispatcher.**find_handlers**(*req*, *paths*)

circuits.web.dispatchers.dispatcher.**resolve_methods**(*parts*)

circuits.web.dispatchers.dispatcher.**resolve_path**(*paths*, *parts*)

**circuits.web.dispatchers.jsonrpc module** JSON RPC

This module implements a JSON RPC dispatcher that translates incoming RPC calls over JSON into RPC events.

**class** circuits.web.dispatchers.jsonrpc.**JSONRPC**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.web.dispatchers.jsonrpc.**rpc**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

RPC Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'rpc'**

### circuits.web.dispatchers.static module

Static

This modStatic implements a Static dispatcher used to serve up static resources and an optional apache-style directory listing.

**class** circuits.web.dispatchers.static.**Static**(*path=None*, *docroot=None*, *defaults=('index.html', 'index.xhtml')*, *dirlisting=False*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

### circuits.web.dispatchers.virtualhosts module

VirtualHost

This module implements a virtual host dispatcher that sends requests for configured virtual hosts to different dispatchers.

**class** circuits.web.dispatchers.virtualhosts.**VirtualHosts**(*domains*)

Bases: *circuits.core.components.BaseComponent*

Forward to anotehr Dispatcher based on the Host header.

This can be useful when running multiple sites within one server. It allows several domains to point to different parts of a single website structure. For example: - http://www.domain.example -> / - http://www.domain2.example -> /domain2 - http://www.domain2.example:443 -> /secure

**Parameters** **domains** (*dict*) – a dict of {host header value: virtual prefix} pairs.

The incoming "Host" request header is looked up in this dict, and, if a match is found, the corresponding "virtual prefix" value will be prepended to the URL path before passing the request onto the next dispatcher.

Note that you often need separate entries for "example.com" and "www.example.com". In addition, "Host" headers may contain the port number.

**channel = 'web'**

**circuits.web.dispatchers.xmlrpc module** XML RPC

This module implements a XML RPC dispatcher that translates incoming RPC calls over XML into RPC events.

**class** circuits.web.dispatchers.xmlrpc.**XMLRPC**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.web.dispatchers.xmlrpc.**rpc**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

rpc Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'rpc'**

**Module contents** Dispatchers

This package contains various circuits.web dispatchers By default a `circuits.web.Server` Component uses the `dispatcher.Dispatcher`

### circuits.web.parsers package

#### Submodules

#### circuits.web.parsers.http module

**class** `circuits.web.parsers.http.`**`HttpParser`**(*kind=2*, *decompress=False*)

Bases: `object`

**`execute`**(*data*, *length*)

**`get_headers`**()

**`get_method`**()

**`get_path`**()

**`get_query_string`**()

**`get_scheme`**()

**`get_status_code`**()

**`get_url`**()

**`get_version`**()

**`is_chunked`**()
return True if Transfer-Encoding header value is chunked

**`is_headers_complete`**()
return True if all headers have been parsed.

**`is_message_begin`**()
return True if the parsing start

**`is_message_complete`**()
return True if the parsing is done (we get EOF)

**`is_partial_body`**()
return True if a chunk of body have been parsed

**`is_upgrade`**()
Do we get upgrade header in the request. Useful for websockets

**`recv_body`**()
return last chunk of the parsed body

**`recv_body_into`**(*barray*)
Receive the last chunk of the parsed bodyand store the data in a buffer rather than creating a new string.

**`should_keep_alive`**()
return True if the connection should be kept alive

**exception** `circuits.web.parsers.http.`**`InvalidChunkSize`**

Bases: `exceptions.Exception`

error raised when we parse an invalid chunk size

**exception** circuits.web.parsers.http.**InvalidHeader**
Bases: exceptions.Exception

error raised on invalid header

**exception** circuits.web.parsers.http.**InvalidRequestLine**
Bases: exceptions.Exception

error raised when first line is invalid

#### circuits.web.parsers.multipart module

**Parser for multipart/form-data** This module provides a parser for the multipart/form-data format. It can read from a file, a socket or a WSGI environment. The parser can be used to replace cgi.FieldStorage (without the bugs) and works with Python 2.5+ and 3.x (2to3).

#### Licence (MIT)

Copyright (c) 2010, Marcel Hellkamp. Inspired by the Werkzeug library: http://werkzeug.pocoo.org/

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**class** circuits.web.parsers.multipart.**MultiDict**(**a*, ***k*)
Bases: _abcoll.MutableMapping

A dict that remembers old values for each key

**append**(*key*, *value*)

**get**(*key*, *default=None*, *index=-1*)

**getall**(*key*)

**iterallitems**()

**keys**()

**replace**(*key*, *value*)

**exception** circuits.web.parsers.multipart.**MultipartError**
Bases: exceptions.ValueError

**class** circuits.web.parsers.multipart.**MultipartParser**(*stream*, *boundary*, *content_length=-1*, *disk_limit=1073741824*, *mem_limit=1048576*, *memfile_limit=262144*, *buffer_size=65536*, *charset='latin1'*)

Bases: object

Parse a multipart/form-data byte stream. This object is an iterator over the parts of the message.

**Parameters**

- **stream** – A file-like stream. Must implement .read(size).
- **boundary** – The multipart boundary as a byte string.
- **content_length** – The maximum number of bytes to read.

**get**(*name*, *default=None*)
Return the first part with that name or a default value (None).

**get_all**(*name*)
Return a list of parts with that name.

**parts**()
Returns a list with all parts of the multipart message.

**class** circuits.web.parsers.multipart.**MultipartPart**(*buffer_size=65536*, *memfile_limit=262144*, *charset='latin1'*)

Bases: object

**feed**(*line*, *nl=''*)

**finish_header**()

**is_buffered**()
Return true if the data is fully buffered in memory.

**save_as**(*path*)

**value**
Data decoded with the specified charset

**write_body**(*line*, *nl*)

**write_header**(*line*, *nl*)

circuits.web.parsers.multipart.**copy_file**(*stream*, *target*, *maxread=-1*, *buffer_size=32*)
Read from :stream and write to :target until :maxread or EOF.

circuits.web.parsers.multipart.**header_quote**(*val*)

circuits.web.parsers.multipart.**header_unquote**(*val*, *filename=False*)

circuits.web.parsers.multipart.**parse_form_data**(*environ*, *charset='utf8'*, *strict=False*, ***kw*)
Parse form data from an environ dict and return a (forms, files) tuple. Both tuple values are dictionaries with the form-field name as a key (text_type) and lists as values (multiple values per key are possible). The forms-dictionary contains form-field values as text_type strings. The files-dictionary contains *MultipartPart* instances, either because the form-field was a file-upload or the value is to big to fit into memory limits.

**Parameters**

- **environ** – An WSGI environment dict.

- **charset** – The charset to use if unsure. (default: utf8)
- **strict** – If True, raise *MultipartError* on any parsing errors. These are silently ignored by default.

circuits.web.parsers.multipart.**parse_options_header**(*header*, *options=None*)

circuits.web.parsers.multipart.**tob**(*data*, *enc='utf8'*)

**circuits.web.parsers.querystring module**

**class** circuits.web.parsers.querystring.**QueryStringParser**(*data*)

Bases: object

**parse**(*key*, *value*)

**process**(*pair*)

**tokens**(*key*)

**class** circuits.web.parsers.querystring.**QueryStringToken**

Bases: object

**ARRAY = 'ARRAY'**

**KEY = 'KEY'**

**OBJECT = 'OBJECT'**

**Module contents** circuits.web parsers

### circuits.web.websockets package

**Submodules**

**circuits.web.websockets.client module**

**class** circuits.web.websockets.client.**WebSocketClient**(*url*, *channel='wsclient'*, *wschannel='ws'*, *headers=None*)

Bases: *circuits.core.components.BaseComponent*

An RFC 6455 compliant WebSocket client component. Upon receiving a circuits.web.client.Connect event, the component tries to establish the connection to the server in a two stage process. First, a *circuits.net.events.connect* event is sent to a child *TCPClient*. When the TCP connection has been established, the HTTP request for opening the WebSocket is sent to the server. A failure in this setup process is signaled by raising an *NotConnected* exception.

When the server accepts the request, the WebSocket connection is established and can be used very much like an ordinary socket by handling *read* events on and sending *write* events to the channel specified as the wschannel parameter of the constructor. Firing a *close* event on that channel closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

**Parameters**

- **url** – the URL to connect to.
- **channel** – the channel used by this component
- **wschannel** – the channel used for the actual WebSocket communication (read, write, close events)
- **headers** – additional headers to be passed with the WebSocket setup HTTP request

**channel = 'wsclient'**

**close**()

**connected**

**circuits.web.websockets.dispatcher module**

**class** circuits.web.websockets.dispatcher.**WebSocketsDispatcher**(*path=None*, *wschannel='wsserver'*, **args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

This class implements an RFC 6455 compliant WebSockets dispatcher that handles the WebSockets handshake and upgrades the connection.

The dispatcher listens on its channel for `Request` events and tries to match them with a given path. Upon a match, the request is checked for the proper Opening Handshake information. If successful, the dispatcher confirms the establishment of the connection to the client. Any subsequent data from the client is handled as a WebSocket data frame, decoded and fired as a `Read` event on the `wschannel` passed to the constructor. The data from *write* events on that channel is encoded as data frames and forwarded to the client.

Firing a `Close` event on the `wschannel` closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

**Parameters**

- **path** – the path to handle. Requests that start with this path are considered to be WebSocket Opening Handshakes.
- **wschannel** – the channel on which `read` events from the client will be delivered and where *write* events to the client will be sent to.

**channel = 'web'**

**Module contents** circuits.web websockets

## Submodules

### circuits.web.client module

**class** circuits.web.client.**Client**(*channel='client'*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'client'**

**close**()

**connect**(*event*, *host=None*, *port=None*, *secure=None*)

**connected**

**request**(*method*, *url*, *body=None*, *headers=None*)

**response**

**write**(*data*)

**exception** circuits.web.client.**HTTPException**

Bases: exceptions.Exception

**exception** circuits.web.client.**NotConnected**
Bases: *circuits.web.client.HTTPException*

circuits.web.client.**parse_url**(*url*)

**class** circuits.web.client.**request**(*method*, *path*, *body=None*, *headers=None*)
Bases: *circuits.core.events.Event*

request Event

This Event is used to initiate a new request.

**Parameters**

- **method** (*str*) – HTTP Method (PUT, GET, POST, DELETE)
- **url** (*str*) – Request URL

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'request'**

### circuits.web.constants module

Global Constants

This module implements required shared global constants.

### circuits.web.controllers module

Controllers

This module implements ...

**class** circuits.web.controllers.**BaseController**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = '/'**

**expires**(*secs=0*, *force=False*)

**forbidden**(*description=None*)
Return a 403 (Forbidden) response

**Parameters** **description** (*str*) – Message to display

**notfound**(*description=None*)
Return a 404 (Not Found) response

**Parameters** **description** (*str*) – Message to display

**redirect**(*urls*, *code=None*)
Return a 30x (Redirect) response

Redirect to another location specified by urls with an optional custom response code.

**Parameters**

- **urls** (*str or list*) – A single URL or list of URLs
- **code** (*int*) – HTTP Redirect code

**serve_download**(*path*, *name=None*)

**serve_file**(*path*, *type=None*, *disposition=None*, *name=None*)

**uri**

Return the current Request URI

**See also:**

*circuits.web.url.URL*

**class** circuits.web.controllers.**Controller**(**args*, ***kwargs*)
Bases: *circuits.web.controllers.BaseController*

initializes x; see x.__class__.__doc__ for signature

**class** circuits.web.controllers.**ExposeJSONMetaClass**(*name*, *bases*, *dct*)
Bases: type

**class** circuits.web.controllers.**ExposeMetaClass**(*name*, *bases*, *dct*)
Bases: type

**class** circuits.web.controllers.**JSONController**(**args*, ***kwargs*)
Bases: *circuits.web.controllers.BaseController*

initializes x; see x.__class__.__doc__ for signature

circuits.web.controllers.**expose**(**channels*, ***config*)

circuits.web.controllers.**exposeJSON**(**channels*, ***config*)

### circuits.web.errors module

Errors

This module implements a set of standard HTTP Errors.

**class** circuits.web.errors.**forbidden**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Forbidden error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 403**

**name = 'forbidden'**

**class** circuits.web.errors.**httperror**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.core.events.Event*

An event for signaling an HTTP error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 500**

**description = ''**

**name = 'httperror'**

**sanitize**()

**class** circuits.web.errors.**notfound**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Not Fouond error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 404**

**name = 'notfound'**

**class** circuits.web.errors.**redirect**(*request*, *response*, *urls*, *code=None*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Redirect response

The constructor creates a new instance and modifies the *response* argument to reflect a redirect response to the given *url*.

**name = 'redirect'**

**class** circuits.web.errors.**unauthorized**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Unauthorized error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 401**

**name = 'unauthorized'**

### circuits.web.events module

Events

This module implements the necessary Events needed.

**class** circuits.web.events.**request**(*Event*) → request Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'request'**

**success = True**

**class** `circuits.web.events.`**response**(*Event*) → response Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'response'**

**success = True**

**class** circuits.web.events.**stream**(*Event*) → stream Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'stream'**

**success = True**

**class** circuits.web.events.**terminate**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

terminate Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'terminate'**

### circuits.web.exceptions module

Exceptions

This module implements a set of standard HTTP Errors as Python Exceptions.

Note: This code is mostly borrowed from werkzeug and adapted for circuits.web

**exception** `circuits.web.exceptions.`**Redirect**(*urls*, *status=None*)
Bases: *circuits.web.exceptions.HTTPException*

**code = 303**

**exception** `circuits.web.exceptions.`**RequestTimeout**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*408 Request Timeout*

Raise to signalize a timeout.

**code = 408**

**description = "<p>The server closed the network connection because the browser didn't finish the request within the**

**exception** circuits.web.exceptions.**RequestEntityTooLarge**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*413 Request Entity Too Large*

The status code one should return if the data submitted exceeded a given limit.

**code = 413**

**description = ' The data value transmitted exceeds the capacity limit. '**

**exception** circuits.web.exceptions.**ServiceUnavailable**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*503 Service Unavailable*

Status code you should return if a service is temporarily unavailable.

**code = 503**

**description = ' The server is temporarily unable to service your request due to maintenance downtime or capacity**

**exception** circuits.web.exceptions.**UnsupportedMediaType**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*415 Unsupported Media Type*

The status code returned if the server is unable to handle the media type the client transmitted.

**code = 415**

**description = ' The server does not support the media type transmitted in the request. '**

**exception** circuits.web.exceptions.**RequestURITooLarge**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*414 Request URI Too Large*

Like *413* but for too long URLs.

**code = 414**

**description = ' The length of the requested URL exceeds the capacity limit for this server. The request cannot be p**

**exception** circuits.web.exceptions.**BadGateway**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*502 Bad Gateway*

If you do proxying in your application you should return this status code if you received an invalid response from the upstream server it accessed in attempting to fulfill the request.

**code = 502**

**description = ' The proxy server received an invalid response from an upstream server. '**

**exception** circuits.web.exceptions.**HTTPException**(*description=None*, *traceback=None*)

Bases: exceptions.Exception

Baseclass for all HTTP exceptions. This exception can be called by WSGI applications to render a default error page or you can catch the subclasses of it independently and render nicer error messages.

**code = None**

**description = None**

**name**
The status name.

**traceback = True**

**exception** circuits.web.exceptions.**UnicodeError**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

raised by the request functions if they were unable to decode the incoming data properly.

**exception** circuits.web.exceptions.**NotAcceptable**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*406 Not Acceptable*

Raise if the server can't return any content conforming to the *Accept* headers of the client.

**code = 406**

**description = ' The resource identified by the request is only capable of generating response entities which have co**

**exception** circuits.web.exceptions.**BadRequest**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*400 Bad Request*

Raise if the browser sends something to the application the application or server cannot handle.

**code = 400**

**description = ' The browser (or proxy) sent a request that this server could not understand. '**

**exception** circuits.web.exceptions.**RangeUnsatisfiable**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*416 Range Unsatisfiable*

The status code returned if the server is unable to satisfy the request range

**code = 416**

**description = ' The server cannot satisfy the request range(s). '**

**exception** circuits.web.exceptions.**Unauthorized**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*401 Unauthorized*

Raise if the user is not authorized. Also used if you want to use HTTP basic auth.

**code = 401**

**description = " The server could not verify that you are authorized to access the URL requested. You either suppli**

**exception** circuits.web.exceptions.**Gone**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*410 Gone*

Raise if a resource existed previously and went away without new location.

**code = 410**

**description = ' The requested URL is no longer available on this server and there is no forwarding address. <p**

**exception** circuits.web.exceptions.**PreconditionFailed**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*412 Precondition Failed*

Status code used in combination with `If-Match`, `If-None-Match`, or `If-Unmodified-Since`.

**code = 412**

**description = ' The precondition on the request for the URL failed positive evaluation. '**

**exception** circuits.web.exceptions.**InternalServerError**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*500 Internal Server Error*

Raise if an internal server error occurred. This is a good fallback if an unknown error occurred in the dispatcher.

**code = 500**

**description = ' The server encountered an internal error and was unable to complete your request. Either the serv**

**exception** circuits.web.exceptions.**MethodNotAllowed**(*method*, *description=None*)

Bases: *circuits.web.exceptions.HTTPException*

*405 Method Not Allowed*

Raise if the server used a method the resource does not handle. For example *POST* if the resource is view only. Especially useful for REST.

The first argument for this exception should be a list of allowed methods. Strictly speaking the response would be invalid if you don't provide valid methods in the header which you can do with that list.

**code = 405**

**exception** circuits.web.exceptions.**NotFound**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*404 Not Found*

Raise if a resource does not exist and never existed.

**code = 404**

**description = ' The requested URL was not found on the server. If you entered the URL manually please c**

**exception** circuits.web.exceptions.**LengthRequired**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*411 Length Required*

Raise if the browser submitted data but no `Content-Length` header which is required for the kind of processing the server does.

**code = 411**

**description = ' A request with this method requires a valid <code>Content-Length</code> header. '**

**exception** circuits.web.exceptions.**Forbidden**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*403 Forbidden*

Raise if the user doesn't have the permission for the requested resource but was authenticated.

**code = 403**

**description = " You don't have the permission to access the requested resource. It is either read-protected or not r**

**exception** circuits.web.exceptions.**NotImplemented**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*501 Not Implemented*

Raise if the application does not support the action requested by the browser.

**code = 501**

**description = ' The server does not support the action requested by the browser. '**

### circuits.web.headers module

Headers Support

This module implements support for parsing and handling headers.

**class** circuits.web.headers.**AcceptElement**(*value*, *params=None*)
Bases: *circuits.web.headers.HeaderElement*

An element (with parameters) from an Accept* header's element list.

AcceptElement objects are comparable; the more-preferred object will be "less than" the less-preferred object. They are also therefore sortable; if you sort a list of AcceptElement objects, they will be listed in priority order; the most preferred value will be first. Yes, it should have been the other way around, but it's too late to fix now.

**classmethod from_str**(*elementstr*)

**qvalue**
The qvalue, or priority, of this value.

**class** circuits.web.headers.**CaseInsensitiveDict**(**args*, ***kwargs*)
Bases: dict

A case-insensitive dict subclass.

Each key is changed on entry to str(key).title().

**classmethod fromkeys**(*seq*, *value=None*)

**get**(*key*, *default=None*)

**pop**(*key*, *default=None*)

**setdefault**(*key*, *x=None*)

**update**(*E*)

**class** circuits.web.headers.**HeaderElement**(*value*, *params=None*)
Bases: object

An element (with parameters) from an HTTP header's element list.

**classmethod from_str**(*elementstr*)
Construct an instance from a string of the form 'token;key=val'.

**static parse**(*elementstr*)
Transform 'token;key=val' to ('token', {'key': 'val'}).

**class** circuits.web.headers.**Headers**(**args*, ***kwargs*)
Bases: *circuits.web.headers.CaseInsensitiveDict*

This class implements a storage for headers as key value pairs. The underlying model of a case insensitive dict matches the requirements for headers quite well, because usually header keys are unique. If several values may be associated with a header key, most HTTP headers represent the values as an enumeration using a comma as item separator.

There is, however one exception (currently) to this rule. In order to set several cookies, there should be multiple headers with the same key, each setting one cookie ("Set-Cookie: some_cookie").

This is modeled by having either a string (common case) or a list (cookie case) as value in the underlying dict. In order to allow easy iteration over all headers as they appear in the HTTP request, the items() method expands associated lists of values. So if you have { "Set-Cookie": [ "cookie1", "cookie2" ] }, the items() method returns the two pairs ("Set-Cookie", "cookie1") and ("Set-Cookie", "cookie2"). This is convenient for most use cases. The only drawback is that len(keys()) is not equal to len(items()) for this specialized dict.

**add_header**(*_name*, *_value*, ***_params*)
Extended header setting.

_name is the header field to add. keyword arguments can be used to set additional parameters for the header field, with underscores converted to dashes. Normally the parameter will be added as key="value" unless value is None, in which case only the key will be added.

Example:

h.add_header('content-disposition', 'attachment', filename='bud.gif')

Note that unlike the corresponding 'email.Message' method, this does *not* handle '(charset, language, value)' tuples: all values must be strings or None.

**append**(*key*, *value*)
If a header with the given name already exists, the value is normally appended to the existing value separated by a comma.

If, however, the already existing entry associated key with a value of type list (as is the case for "Set-Cookie"), the new value is appended to that list.

**elements**(*key*)
Return a sorted list of HeaderElements for the given header.

**get_all**(*name*)
Return a list of all the values for the named field.

**items**()

circuits.web.headers.**header_elements**(*fieldname*, *fieldvalue*)
Return a sorted HeaderElement list.

Returns a sorted HeaderElement list from a comma-separated header string.

### circuits.web.http module

Hyper Text Transfer Protocol

This module implements the server side Hyper Text Transfer Protocol or commonly known as HTTP.

**class** circuits.web.http.**HTTP**(*server*, *encoding='utf-8'*, *channel='web'*)
Bases: *circuits.core.components.BaseComponent*

HTTP Protocol Component

Implements the HTTP server protocol and parses and processes incoming HTTP messages, creating and sending an appropriate response.

The component handles `Read` events on its channel and collects the associated data until a complete HTTP request has been received. It parses the request's content and puts it in a *Request* object and creates a corresponding *Response* object. Then it emits a `Request` event with these objects as arguments.

The component defines several handlers that send a response back to the client.

**base**

**channel = 'web'**

**protocol**

**scheme**

**version**

### circuits.web.loggers module

Logger Component

This module implements Logger Components.

**class** circuits.web.loggers.**Logger**(*file=None*, *logger=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**format = '%(h)s %(l)s %(u)s %(t)s "%(r)s" %(s)s %(b)s "%(f)s" "%(a)s"'**

**log**(*response*)

**log_response**(*response_event*, *value*)

circuits.web.loggers.**formattime**()

### circuits.web.main module

Main

circutis.web Web Server and Testing Tool.

**class** circuits.web.main.**Authentication**(*channel='web'*, *realm=None*, *passwd=None*)
Bases: *circuits.core.components.Component*

**channel = 'web'**

**realm = 'Secure Area'**

**request**(*event*, *request*, *response*)

**users = {'admin': '21232f297a57a5a743894a0e4a801fc3'}**

**class** circuits.web.main.**HelloWorld**(**args*, ***kwargs*)
Bases: *circuits.core.components.Component*

initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**request**(*request*, *response*)

**class** circuits.web.main.**Root**(**args*, ***kwargs*)
Bases: *circuits.web.controllers.Controller*

initializes x; see x.__class__.__doc__ for signature

**hello**(*event*, **args*, ***kwargs*)

circuits.web.main.**main**()

circuits.web.main.**parse_bind**(*bind*)

circuits.web.main.**parse_options**()

circuits.web.main.**select_poller**(*poller*)

### circuits.web.processors module

circuits.web.processors.**process**(*request*, *params*)

circuits.web.processors.**process_multipart**(*request*, *params*)

circuits.web.processors.**process_urlencoded**(*request*, *params*, *encoding='utf-8'*)

### circuits.web.servers module

Web Servers

This module implements the several Web Server components.

**class** circuits.web.servers.**BaseServer**(*bind*, *encoding='utf-8'*, *secure=False*, *certfile=None*, *channel='web'*, *display_banner=True*)
Bases: *circuits.core.components.BaseComponent*

Create a Base Web Server

Create a Base Web Server (HTTP) bound to the IP Address / Port or UNIX Socket specified by the 'bind' parameter.

**Variables** ***server*** – Reference to underlying Server Component

**Parameters** **bind** (*Instance of int, list, tuple or str*) – IP Address / Port or UNIX Socket to bind to.

The 'bind' parameter is quite flexible with what valid values it accepts.

If an int is passed, a TCPServer will be created. The Server will be bound to the Port given by the 'bind' argument and the bound interface will default (normally to "0.0.0.0").

If a list or tuple is passed, a TCPServer will be created. The Server will be bound to the Port given by the 2nd item in the 'bind' argument and the bound interface will be the 1st item.

If a str is passed and it contains the ':' character, this is assumed to be a request to bind to an IP Address / Port. A TCpServer will thus be created and the IP Address and Port will be determined by splitting the string given by the 'bind' argument.

Otherwise if a str is passed and it does not contain the ':' character, a file path is assumed and a UNIXServer is created and bound to the file given by the 'bind' argument.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**display_banner**

**host**

**port**

**secure**

**class** circuits.web.servers.**FakeSock**

**getpeername**()

**class** circuits.web.servers.**Server**(*bind*, ***kwargs*)
Bases: *circuits.web.servers.BaseServer*

Create a Web Server

Create a Web Server (HTTP) complete with the default Dispatcher to parse requests and posted form data dispatching to appropriate Controller(s).

See: circuits.web.servers.BaseServer

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**class** circuits.web.servers.**StdinServer**(*encoding='utf-8'*, *channel='web'*)
Bases: *circuits.core.components.BaseComponent*

**channel** = 'web'

**host**

**port**

**read**(*data*)

**secure**

**write**(*sock*, *data*)

### circuits.web.sessions module

Session Components

This module implements Session Components that can be used to store and access persistent information.

**class** circuits.web.sessions.**Sessions**(*name='circuits.session'*, *channel='web'*)
Bases: *circuits.core.components.Component*

**channel** = 'web'

**load**(*sid*)

**request**(*request*, *response*)

**save**(*sid*, *data*)
Save User Session Data for sid

circuits.web.sessions.**create_session**(*request*)
Create a unique session id from the request

Returns a unique session using `uuid4()` and a `sha1()` hash of the users IP Address and User Agent in the form of `sid/who`.

circuits.web.sessions.**verify_session**(*request*, *sid*)
Verify a User's Session

This verifies the User's Session by verifying the SHA1 Hash of the User's IP Address and User-Agent match the provided Session ID.

circuits.web.sessions.**who**(*request*, *encoding='utf-8'*)
Create a SHA1 Hash of the User's IP Address and User-Agent

### circuits.web.tools module

Tools

This module implements tools used throughout circuits.web. These tools can also be used within Controlelrs and request handlers.

**class** circuits.web.tools.**ReverseProxy**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**headers = ('X-Real-IP', 'X-Forwarded-For')**

**init**(*headers=None*)
Web Component for identifying the original client IP when a reverse proxy is used

**Parameters** **headers** – List of HTTP headers to read the original client IP

circuits.web.tools.**basic_auth**(*request*, *response*, *realm*, *users*, *encrypt=None*)
Perform Basic Authentication

If auth fails, returns an Unauthorized error with a basic authentication header.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.
- **encrypt** (*callable*) – Callable used to encrypt the password returned from the user-agent. if None it defaults to a md5 encryption.

circuits.web.tools.**check_auth**(*request*, *response*, *realm*, *users*, *encrypt=None*)
Check Authentication

If an Authorization header contains credentials, return True, else False.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.
- **encrypt** (*callable*) – Callable used to encrypt the password returned from the user-agent. if None it defaults to a md5 encryption.

circuits.web.tools.**digest_auth**(*request*, *response*, *realm*, *users*)
Perform Digest Authentication

If auth fails, raise 401 with a digest authentication header.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.

circuits.web.tools.**expires**(*request*, *response*, *secs=0*, *force=False*)
Tool for influencing cache mechanisms using the 'Expires' header.

'secs' must be either an int or a datetime.timedelta, and indicates the number of seconds between response.time and when the response should expire. The 'Expires' header will be set to (response.time + secs).

If 'secs' is zero, the 'Expires' header is set one year in the past, and the following "cache prevention" headers are also set: - 'Pragma': 'no-cache' - 'Cache-Control': 'no-cache, must-revalidate'

If 'force' is False (the default), the following headers are checked: 'Etag', 'Last-Modified', 'Age', 'Expires'. If any are already present, none of the above response headers are set.

circuits.web.tools.**gzip**(*response*, *level=4*, *mime_types=('text/html', 'text/plain')*)
Try to gzip the response body if Content-Type in mime_types.

response.headers['Content-Type'] must be set to one of the values in the mime_types arg before calling this function.

**No compression is performed if any of the following hold:**

- The client sends no Accept-Encoding request header
- No 'gzip' or 'x-gzip' is present in the Accept-Encoding header
- No 'gzip' or 'x-gzip' with a qvalue > 0 is present
- The 'identity' value is given with a qvalue > 0.

circuits.web.tools.**serve_download**(*request*, *response*, *path*, *name=None*)
Serve 'path' as an application/x-download attachment.

circuits.web.tools.**serve_file**(*request*, *response*, *path*, *type=None*, *disposition=None*, *name=None*)
Set status, headers, and body in order to serve the given file.

The Content-Type header will be set to the type arg, if provided. If not provided, the Content-Type will be guessed by the file extension of the 'path' argument.

If disposition is not None, the Content-Disposition header will be set to "<disposition>; filename=<name>". If name is None, it will be set to the basename of path. If disposition is None, no Content-Disposition header will be written.

circuits.web.tools.**validate_etags**(*request*, *response*, *autotags=False*)
Validate the current ETag against If-Match, If-None-Match headers.

If autotags is True, an ETag response-header value will be provided from an MD5 hash of the response body (unless some other code has already provided an ETag header). If False (the default), the ETag will not be automatic.

WARNING: the autotags feature is not designed for URL's which allow methods other than GET. For example, if a POST to the same URL returns no content, the automatic ETag will be incorrect, breaking a fundamental use for entity tags in a possibly destructive fashion. Likewise, if you raise 304 Not Modified, the response body will be empty, the ETag hash will be incorrect, and your application will break. See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.24

circuits.web.tools.**validate_since**(*request*, *response*)
Validate the current Last-Modified against If-Modified-Since headers.

If no code has set the Last-Modified response header, then no validation will be performed.

### circuits.web.url module

This is a module for dealing with urls. In particular, sanitizing them.

**class** circuits.web.url.**URL**(*scheme*, *host*, *port*, *path*, *params=''*, *query=''*, *fragment=''*)
Bases: object

**For more information on how and what we parse / sanitize:** http://tools.ietf.org/html/rfc1808.html

**The more up-to-date RFC is this one:** http://www.ietf.org/rfc/rfc3986.txt

**absolute**()
Return True if this is a fully-qualified URL with a hostname and everything

**abspath**()
Clear out any '..' and excessive slashes from the path

**canonical**()
Canonicalize this url. This includes reordering parameters and args to have a consistent ordering

**defrag**()
Remove the fragment from this url

**deparam**(*params=None*)
Strip any of the provided parameters out of the url

**encode**(*encoding*)
Return the url in an arbitrary encoding

**equiv**(*other*)
Return true if this url is equivalent to another

**escape**()
Make sure that the path is correctly escaped

**lower**()
Lowercase the hostname

**classmethod parse**(*url*, *encoding*)
Parse the provided url, and return a URL instance

**punycode**()
Convert to punycode hostname

**relative**(*path*, *encoding='utf-8'*)
Evaluate the new path relative to the current url

**sanitize**()
A shortcut to abspath, escape and lowercase

**unescape**()
Unescape the path

**unicode**()
Return a unicode version of this url

**unpunycode**()
Convert to an unpunycoded hostname

**utf8**()
Return a utf-8 version of this url

circuits.web.url.**parse_url**(*url*, *encoding='utf-8'*)
Parse the provided url string and return an URL object

### circuits.web.utils module

Utilities

This module implements utility functions.

circuits.web.utils.**average**(*xs*)

circuits.web.utils.**compress**(*body*, *compress_level*)
Compress 'body' at the given compress_level.

circuits.web.utils.**dictform**(*form*)

circuits.web.utils.**get_ranges**(*headervalue*, *content_length*)
Return a list of (start, stop) indices from a Range header, or None.

Each (start, stop) tuple will be composed of two ints, which are suitable for use in a slicing operation. That is, the header "Range: bytes=3-6", if applied against a Python string, is requesting resource[3:7]. This function will return the list [(3, 7)].

If this function returns an empty list, you should return HTTP 416.

circuits.web.utils.**parse_body**(*request*, *response*, *params*)

circuits.web.utils.**parse_qs**(*query_string*) → dict
Build a params dictionary from a query_string. If keep_blank_values is True (the default), keep values that are blank.

circuits.web.utils.**stddev**(*xs*)

circuits.web.utils.**variance**(*xs*)

### circuits.web.wrappers module

Request/Response Wrappers

This module implements the Request and Response objects.

**class** circuits.web.wrappers.**Body**
Bases: object

Response Body

**class** circuits.web.wrappers.**HTTPStatus**(*status=200*, *reason=None*)
Bases: object

**reason**

**status**

**class** circuits.web.wrappers.**Host**(*ip*, *port*, *name=None*)
Bases: object

An internet address.

name should be the client's host name. If not available (because no DNS lookup is performed), the IP address should be used instead.

**ip = '0.0.0.0'**

**name = 'unknown.tld'**

**port = 80**

**class** circuits.web.wrappers.**Request**(*sock*, *method='GET'*, *scheme='http'*, *path='/'*, *protocol=(1, 1)*, *qs=''*, *headers=None*, *server=None*)

Bases: object

Creates a new Request object to hold information about a request.

**Parameters**

- **sock** (*socket.socket*) – The socket object of the request.
- **method** (*str*) – The requested method.
- **scheme** (*str*) – The requested scheme.
- **path** (*str*) – The requested path.
- **protocol** (*str*) – The requested protocol.
- **qs** (*str*) – The query string of the request.

initializes x; see x.__class__.__doc__ for signature

**handled = False**

**host = ''**

**index = None**

**local = Host('127.0.0.1', 80, '127.0.0.1')**

**login = None**

**protocol = (1, 1)**

**remote = Host('', 0, '')**

**scheme = 'http'**

**script_name = ''**

**server = None**

**class** circuits.web.wrappers.**Response**(*sock*, *request*) → new Response object

Bases: object

A Response object that holds the response to send back to the client. This ensure that the correct data is sent in the correct order.

initializes x; see x.__class__.__doc__ for signature

**body**

Response Body

**chunked = False**

**close = False**

**done = False**

**prepare**()

**status**

Response Status

**stream = False**

**class** circuits.web.wrappers.**Status**

Bases: object

Response Status

circuits.web.wrappers.**file_generator**(*input*, *chunkSize=4096*)

#### circuits.web.wsgi module

WSGI Components

This module implements WSGI Components.

**class** circuits.web.wsgi.**Application**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**getRequestResponse**(*environ*)

**headerNames = {'CONTENT_LENGTH': 'Content-Length', 'REMOTE_HOST': 'Remote-Host', 'CONTENT_TYPE':**

**host**

**init**()

**port**

**response**(*event*, *response*)

**secure**

**translateHeaders**(*environ*)

**class** circuits.web.wsgi.**Gateway**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**init**(*apps*)

circuits.web.wsgi.**create_environ**(*errors*, *path*, *req*)

### Module contents

Circuits Library - Web

circuits.web contains the circuits full stack web server that is HTTP and WSGI compliant.

## 2.5.42 circuits.web.client module

**class** circuits.web.client.**Client**(*channel='client'*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'client'**

**close**()

**connect**(*event*, *host=None*, *port=None*, *secure=None*)

**connected**

**request**(*method*, *url*, *body=None*, *headers=None*)

**response**

**write**(*data*)

**exception** circuits.web.client.**HTTPException**
Bases: exceptions.Exception

**exception** circuits.web.client.**NotConnected**
Bases: *circuits.web.client.HTTPException*

circuits.web.client.**parse_url**(*url*)

**class** circuits.web.client.**request**(*method*, *path*, *body=None*, *headers=None*)
Bases: *circuits.core.events.Event*

request Event

This Event is used to initiate a new request.

**Parameters**

- **method** (*str*) – HTTP Method (PUT, GET, POST, DELETE)
- **url** (*str*) – Request URL

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**name = 'request'**

### 2.5.43 circuits.web.constants module

Global Constants

This module implements required shared global constants.

### 2.5.44 circuits.web.controllers module

Controllers

This module implements ...

**class** circuits.web.controllers.**BaseController**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = '/'**

**expires**(*secs=0*, *force=False*)

**forbidden**(*description=None*)
Return a 403 (Forbidden) response

**Parameters** **description** (*str*) – Message to display

**notfound**(*description=None*)
Return a 404 (Not Found) response

**Parameters** **description** (*str*) – Message to display

**redirect**(*urls*, *code=None*)
Return a 30x (Redirect) response

Redirect to another location specified by urls with an optional custom response code.

**Parameters**

- **urls** (*str or list*) – A single URL or list of URLs
- **code** (*int*) – HTTP Redirect code

**serve_download**(*path*, *name=None*)

**serve_file**(*path*, *type=None*, *disposition=None*, *name=None*)

**uri**

Return the current Request URI

**See also:**

*circuits.web.url.URL*

**class** circuits.web.controllers.**Controller**(**args*, ***kwargs*)
Bases: *circuits.web.controllers.BaseController*

initializes x; see x.__class__.__doc__ for signature

**class** circuits.web.controllers.**ExposeJSONMetaClass**(*name*, *bases*, *dct*)
Bases: type

**class** circuits.web.controllers.**ExposeMetaClass**(*name*, *bases*, *dct*)
Bases: type

**class** circuits.web.controllers.**JSONController**(**args*, ***kwargs*)
Bases: *circuits.web.controllers.BaseController*

initializes x; see x.__class__.__doc__ for signature

circuits.web.controllers.**expose**(**channels*, ***config*)

circuits.web.controllers.**exposeJSON**(**channels*, ***config*)

## 2.5.45 circuits.web.dispatchers package

### Submodules

#### circuits.web.dispatchers.dispatcher module

Dispatcher

This module implements a basic URL to Channel dispatcher. This is the default dispatcher used by circuits.web

**class** circuits.web.dispatchers.dispatcher.**Dispatcher**(***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

circuits.web.dispatchers.dispatcher.**find_handlers**(*req*, *paths*)

circuits.web.dispatchers.dispatcher.**resolve_methods**(*parts*)

circuits.web.dispatchers.dispatcher.**resolve_path**(*paths*, *parts*)

### circuits.web.dispatchers.jsonrpc module

JSON RPC

This module implements a JSON RPC dispatcher that translates incoming RPC calls over JSON into RPC events.

**class** circuits.web.dispatchers.jsonrpc.**JSONRPC**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.web.dispatchers.jsonrpc.**rpc**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

RPC Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'rpc'**

### circuits.web.dispatchers.static module

Static

This modStatic implements a Static dispatcher used to serve up static resources and an optional apache-style directory listing.

**class** circuits.web.dispatchers.static.**Static**(*path=None*, *docroot=None*, *defaults=('index.html', 'index.xhtml')*, *dirlisting=False*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

### circuits.web.dispatchers.virtualhosts module

VirtualHost

This module implements a virtual host dispatcher that sends requests for configured virtual hosts to different dispatchers.

**class** circuits.web.dispatchers.virtualhosts.**VirtualHosts**(*domains*)

Bases: *circuits.core.components.BaseComponent*

Forward to anotehr Dispatcher based on the Host header.

This can be useful when running multiple sites within one server. It allows several domains to point to different parts of a single website structure. For example: - http://www.domain.example -> / - http://www.domain2.example -> /domain2 - http://www.domain2.example:443 -> /secure

**Parameters** **domains** (*dict*) – a dict of {host header value: virtual prefix} pairs.

The incoming "Host" request header is looked up in this dict, and, if a match is found, the corresponding "virtual prefix" value will be prepended to the URL path before passing the request onto the next dispatcher.

Note that you often need separate entries for "example.com" and "www.example.com". In addition, "Host" headers may contain the port number.

**channel = 'web'**

### circuits.web.dispatchers.xmlrpc module

XML RPC

This module implements a XML RPC dispatcher that translates incoming RPC calls over XML into RPC events.

**class** circuits.web.dispatchers.xmlrpc.**XMLRPC**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.web.dispatchers.xmlrpc.**rpc**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

rpc Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'rpc'**

### Module contents

Dispatchers

This package contains various circuits.web dispatchers By default a `circuits.web.Server` Component uses the `dispatcher.Dispatcher`

## 2.5.46 circuits.web.dispatchers.dispatcher module

Dispatcher

This module implements a basic URL to Channel dispatcher. This is the default dispatcher used by circuits.web

**class** `circuits.web.dispatchers.dispatcher.`**Dispatcher**(***kwargs*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

`circuits.web.dispatchers.dispatcher.`**find_handlers**(*req*, *paths*)

`circuits.web.dispatchers.dispatcher.`**resolve_methods**(*parts*)

`circuits.web.dispatchers.dispatcher.`**resolve_path**(*paths*, *parts*)

## 2.5.47 circuits.web.dispatchers.jsonrpc module

JSON RPC

This module implements a JSON RPC dispatcher that translates incoming RPC calls over JSON into RPC events.

**class** circuits.web.dispatchers.jsonrpc.**JSONRPC**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.web.dispatchers.jsonrpc.**rpc**(**args*, ***kwargs*)

Bases: *circuits.core.events.Event*

RPC Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'rpc'**

## 2.5.48 circuits.web.dispatchers.static module

Static

This modStatic implements a Static dispatcher used to serve up static resources and an optional apache-style directory listing.

**class** circuits.web.dispatchers.static.**Static**(*path=None*, *docroot=None*, *defaults=('index.html', 'index.xhtml')*, *dirlisting=False*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

### 2.5.49 circuits.web.dispatchers.virtualhosts module

VirtualHost

This module implements a virtual host dispatcher that sends requests for configured virtual hosts to different dispatchers.

**class** circuits.web.dispatchers.virtualhosts.**VirtualHosts**(*domains*)
Bases: *circuits.core.components.BaseComponent*

Forward to anotehr Dispatcher based on the Host header.

This can be useful when running multiple sites within one server. It allows several domains to point to different parts of a single website structure. For example: - http://www.domain.example -> / - http://www.domain2.example -> /domain2 - http://www.domain2.example:443 -> /secure

**Parameters** **domains** (*dict*) – a dict of {host header value: virtual prefix} pairs.

The incoming "Host" request header is looked up in this dict, and, if a match is found, the corresponding "virtual prefix" value will be prepended to the URL path before passing the request onto the next dispatcher.

Note that you often need separate entries for "example.com" and "www.example.com". In addition, "Host" headers may contain the port number.

**channel = 'web'**

### 2.5.50 circuits.web.dispatchers.xmlrpc module

XML RPC

This module implements a XML RPC dispatcher that translates incoming RPC calls over XML into RPC events.

**class** circuits.web.dispatchers.xmlrpc.**XMLRPC**(*path=None*, *encoding='utf-8'*, *rpc_channel='*'*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**class** circuits.web.dispatchers.xmlrpc.**rpc**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

rpc Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'rpc'**

### 2.5.51 circuits.web.errors module

Errors

This module implements a set of standard HTTP Errors.

**class** `circuits.web.errors.`**forbidden**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Forbidden error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 403**

**name = 'forbidden'**

**class** `circuits.web.errors.`**httperror**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.core.events.Event*

An event for signaling an HTTP error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 500**

**description = ''**

**name = 'httperror'**

**sanitize**()

**class** `circuits.web.errors.`**notfound**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Not Fouond error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 404**

**name = 'notfound'**

**class** circuits.web.errors.**redirect**(*request*, *response*, *urls*, *code=None*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Redirect response

The constructor creates a new instance and modifies the *response* argument to reflect a redirect response to the given *url*.

**name = 'redirect'**

**class** circuits.web.errors.**unauthorized**(*request*, *response*, *code=None*, ***kwargs*)
Bases: *circuits.web.errors.httperror*

An event for signaling the HTTP Unauthorized error

The constructor creates a new instance and modifies the *response* argument to reflect the error.

**code = 401**

**name = 'unauthorized'**

### 2.5.52 circuits.web.events module

Events

This module implements the necessary Events needed.

**class** circuits.web.events.**request**(*Event*) → request Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'request'**

**success = True**

**class** `circuits.web.events.`**response**(*Event*) → response Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'response'**

**success = True**

**class** circuits.web.events.**stream**(*Event*) → stream Event
Bases: *circuits.core.events.Event*

args: request, response

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.
- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.
- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.
- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.
- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.
- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**complete = True**

**failure = True**

**name = 'stream'**

**success = True**

**class** circuits.web.events.**terminate**(**args*, ***kwargs*)
Bases: *circuits.core.events.Event*

terminate Event

An event is a message send to one or more channels. It is eventually dispatched to all components that have handlers for one of the channels and the event type.

All normal arguments and keyword arguments passed to the constructor of an event are passed on to the handler. When declaring a handler, its argument list must therefore match the arguments used for creating the event.

Every event has a *name* attribute that is used for matching the event with the handlers.

**Variables**

- ***channels*** – an optional attribute that may be set before firing the event. If defined (usually as a class variable), the attribute specifies the channels that the event should be delivered to as a tuple. This overrides the default behavior of sending the event to the firing component's channel.

  When an event is fired, the value in this attribute is replaced for the instance with the channels that the event is actually sent to. This information may be used e.g. when the event is passed as a parameter to a handler.

- ***value*** – this is a *circuits.core.values.Value* object that holds the results returned by the handlers invoked for the event.

- ***success*** – if this optional attribute is set to `True`, an associated event `success` (original name with "_success" appended) will automatically be fired when all handlers for the event have been invoked successfully.

- **success_channels** – the success event is, by default, delivered to same channels as the successfully dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

- ***complete*** – if this optional attribute is set to `True`, an associated event `complete` (original name with "_complete" appended) will automatically be fired when all handlers for the event and all events fired by these handlers (recursively) have been invoked successfully.

- **complete_channels** – the complete event is, by default, delivered to same channels as the initially dispatched event itself. This may be overridden by specifying an alternative list of destinations using this attribute.

**name = 'terminate'**

### 2.5.53 circuits.web.exceptions module

Exceptions

This module implements a set of standard HTTP Errors as Python Exceptions.

Note: This code is mostly borrowed from werkzeug and adapted for circuits.web

**exception** `circuits.web.exceptions.`**Redirect**(*urls*, *status=None*)

Bases: *circuits.web.exceptions.HTTPException*

**code = 303**

**exception** `circuits.web.exceptions.`**RequestTimeout**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*408 Request Timeout*

Raise to signalize a timeout.

**code = 408**

**description = "<p>The server closed the network connection because the browser didn't finish the request within the**

**exception** `circuits.web.exceptions.`**RequestEntityTooLarge**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*413 Request Entity Too Large*

The status code one should return if the data submitted exceeded a given limit.

**code = 413**

**description = ' The data value transmitted exceeds the capacity limit. '**

**exception** circuits.web.exceptions.**ServiceUnavailable**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*503 Service Unavailable*

Status code you should return if a service is temporarily unavailable.

**code = 503**

**description = ' The server is temporarily unable to service your request due to maintenance downtime or capacity**

**exception** circuits.web.exceptions.**UnsupportedMediaType**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*415 Unsupported Media Type*

The status code returned if the server is unable to handle the media type the client transmitted.

**code = 415**

**description = ' The server does not support the media type transmitted in the request. '**

**exception** circuits.web.exceptions.**RequestURITooLarge**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*414 Request URI Too Large*

Like *413* but for too long URLs.

**code = 414**

**description = ' The length of the requested URL exceeds the capacity limit for this server. The request cannot be p**

**exception** circuits.web.exceptions.**BadGateway**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*502 Bad Gateway*

If you do proxying in your application you should return this status code if you received an invalid response from the upstream server it accessed in attempting to fulfill the request.

**code = 502**

**description = ' The proxy server received an invalid response from an upstream server. '**

**exception** circuits.web.exceptions.**HTTPException**(*description=None*, *traceback=None*)

Bases: exceptions.Exception

Baseclass for all HTTP exceptions. This exception can be called by WSGI applications to render a default error page or you can catch the subclasses of it independently and render nicer error messages.

**code = None**

**description = None**

**name**

The status name.

**traceback = True**

**exception** circuits.web.exceptions.**UnicodeError**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

raised by the request functions if they were unable to decode the incoming data properly.

**exception** circuits.web.exceptions.**NotAcceptable**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*406 Not Acceptable*

Raise if the server can't return any content conforming to the *Accept* headers of the client.

**code = 406**

**description = ' The resource identified by the request is only capable of generating response entities which have co**

**exception** circuits.web.exceptions.**BadRequest**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*400 Bad Request*

Raise if the browser sends something to the application the application or server cannot handle.

**code = 400**

**description = ' The browser (or proxy) sent a request that this server could not understand. '**

**exception** circuits.web.exceptions.**RangeUnsatisfiable**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*416 Range Unsatisfiable*

The status code returned if the server is unable to satisfy the request range

**code = 416**

**description = ' The server cannot satisfy the request range(s). '**

**exception** circuits.web.exceptions.**Unauthorized**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*401 Unauthorized*

Raise if the user is not authorized. Also used if you want to use HTTP basic auth.

**code = 401**

**description = " The server could not verify that you are authorized to access the URL requested. You either suppli**

**exception** circuits.web.exceptions.**Gone**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*410 Gone*

Raise if a resource existed previously and went away without new location.

**code = 410**

**description = ' The requested URL is no longer available on this server and there is no forwarding address. <p**

**exception** circuits.web.exceptions.**PreconditionFailed**(*description=None*, *traceback=None*)
Bases: *circuits.web.exceptions.HTTPException*

*412 Precondition Failed*

Status code used in combination with `If-Match`, `If-None-Match`, or `If-Unmodified-Since`.

**code = 412**

**description = ' The precondition on the request for the URL failed positive evaluation. '**

**exception** circuits.web.exceptions.**InternalServerError**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*500 Internal Server Error*

Raise if an internal server error occurred. This is a good fallback if an unknown error occurred in the dispatcher.

**code = 500**

**description = ' The server encountered an internal error and was unable to complete your request. Either the serv**

**exception** circuits.web.exceptions.**MethodNotAllowed**(*method*, *description=None*)

Bases: *circuits.web.exceptions.HTTPException*

*405 Method Not Allowed*

Raise if the server used a method the resource does not handle. For example *POST* if the resource is view only. Especially useful for REST.

The first argument for this exception should be a list of allowed methods. Strictly speaking the response would be invalid if you don't provide valid methods in the header which you can do with that list.

**code = 405**

**exception** circuits.web.exceptions.**NotFound**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*404 Not Found*

Raise if a resource does not exist and never existed.

**code = 404**

**description = ' The requested URL was not found on the server. If you entered the URL manually please c**

**exception** circuits.web.exceptions.**LengthRequired**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*411 Length Required*

Raise if the browser submitted data but no `Content-Length` header which is required for the kind of processing the server does.

**code = 411**

**description = ' A request with this method requires a valid <code>Content-Length</code> header. '**

**exception** circuits.web.exceptions.**Forbidden**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*403 Forbidden*

Raise if the user doesn't have the permission for the requested resource but was authenticated.

**code = 403**

**description = " You don't have the permission to access the requested resource. It is either read-protected or not r**

**exception** circuits.web.exceptions.**NotImplemented**(*description=None*, *traceback=None*)

Bases: *circuits.web.exceptions.HTTPException*

*501 Not Implemented*

Raise if the application does not support the action requested by the browser.

**code = 501**

**description = ' The server does not support the action requested by the browser. '**

### 2.5.54 circuits.web.headers module

Headers Support

This module implements support for parsing and handling headers.

**class** circuits.web.headers.**AcceptElement**(*value*, *params=None*)
Bases: *circuits.web.headers.HeaderElement*

An element (with parameters) from an Accept* header's element list.

AcceptElement objects are comparable; the more-preferred object will be "less than" the less-preferred object. They are also therefore sortable; if you sort a list of AcceptElement objects, they will be listed in priority order; the most preferred value will be first. Yes, it should have been the other way around, but it's too late to fix now.

**classmethod from_str**(*elementstr*)

**qvalue**
The qvalue, or priority, of this value.

**class** circuits.web.headers.**CaseInsensitiveDict**(**args*, ***kwargs*)
Bases: dict

A case-insensitive dict subclass.

Each key is changed on entry to str(key).title().

**classmethod fromkeys**(*seq*, *value=None*)

**get**(*key*, *default=None*)

**pop**(*key*, *default=None*)

**setdefault**(*key*, *x=None*)

**update**(*E*)

**class** circuits.web.headers.**HeaderElement**(*value*, *params=None*)
Bases: object

An element (with parameters) from an HTTP header's element list.

**classmethod from_str**(*elementstr*)
Construct an instance from a string of the form 'token;key=val'.

**static parse**(*elementstr*)
Transform 'token;key=val' to ('token', {'key': 'val'}).

**class** circuits.web.headers.**Headers**(**args*, ***kwargs*)
Bases: *circuits.web.headers.CaseInsensitiveDict*

This class implements a storage for headers as key value pairs. The underlying model of a case insensitive dict matches the requirements for headers quite well, because usually header keys are unique. If several values may be associated with a header key, most HTTP headers represent the values as an enumeration using a comma as item separator.

There is, however one exception (currently) to this rule. In order to set several cookies, there should be multiple headers with the same key, each setting one cookie ("Set-Cookie: some_cookie").

This is modeled by having either a string (common case) or a list (cookie case) as value in the underlying dict. In order to allow easy iteration over all headers as they appear in the HTTP request, the items() method expands associated lists of values. So if you have { "Set-Cookie": [ "cookie1", "cookie2" ] }, the items() method returns the two pairs ("Set-Cookie", "cookie1") and ("Set-Cookie", "cookie2"). This is convenient for most use cases. The only drawback is that len(keys()) is not equal to len(items()) for this specialized dict.

**add_header**(*_name*, *_value*, ***_params*)
Extended header setting.

_name is the header field to add. keyword arguments can be used to set additional parameters for the header field, with underscores converted to dashes. Normally the parameter will be added as key="value" unless value is None, in which case only the key will be added.

Example:

h.add_header('content-disposition', 'attachment', filename='bud.gif')

Note that unlike the corresponding 'email.Message' method, this does *not* handle '(charset, language, value)' tuples: all values must be strings or None.

**append**(*key*, *value*)
If a header with the given name already exists, the value is normally appended to the existing value separated by a comma.

If, however, the already existing entry associated key with a value of type list (as is the case for "Set-Cookie"), the new value is appended to that list.

**elements**(*key*)
Return a sorted list of HeaderElements for the given header.

**get_all**(*name*)
Return a list of all the values for the named field.

**items**()

circuits.web.headers.**header_elements**(*fieldname*, *fieldvalue*)
Return a sorted HeaderElement list.

Returns a sorted HeaderElement list from a comma-separated header string.

### 2.5.55 circuits.web.http module

Hyper Text Transfer Protocol

This module implements the server side Hyper Text Transfer Protocol or commonly known as HTTP.

**class** circuits.web.http.**HTTP**(*server*, *encoding='utf-8'*, *channel='web'*)
Bases: *circuits.core.components.BaseComponent*

HTTP Protocol Component

Implements the HTTP server protocol and parses and processes incoming HTTP messages, creating and sending an appropriate response.

The component handles `Read` events on its channel and collects the associated data until a complete HTTP request has been received. It parses the request's content and puts it in a *Request* object and creates a corresponding *Response* object. Then it emits a `Request` event with these objects as arguments.

The component defines several handlers that send a response back to the client.

**base**

**channel = 'web'**

**protocol**

**scheme**

**version**

### 2.5.56 circuits.web.loggers module

Logger Component

This module implements Logger Components.

**class** circuits.web.loggers.**Logger**(*file=None*, *logger=None*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

**channel = 'web'**

**format = '%(h)s %(l)s %(u)s %(t)s "%(r)s" %(s)s %(b)s "%(f)s" "%(a)s"'**

**log**(*response*)

**log_response**(*response_event*, *value*)

circuits.web.loggers.**formattime**()

### 2.5.57 circuits.web.main module

Main

circutis.web Web Server and Testing Tool.

**class** circuits.web.main.**Authentication**(*channel='web'*, *realm=None*, *passwd=None*)
Bases: *circuits.core.components.Component*

**channel = 'web'**

**realm = 'Secure Area'**

**request**(*event*, *request*, *response*)

**users = {'admin': '21232f297a57a5a743894a0e4a801fc3'}**

**class** circuits.web.main.**HelloWorld**(**args*, ***kwargs*)
Bases: *circuits.core.components.Component*

initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**request**(*request*, *response*)

**class** circuits.web.main.**Root**(**args*, ***kwargs*)
Bases: *circuits.web.controllers.Controller*

initializes x; see x.__class__.__doc__ for signature

**hello**(*event*, **args*, ***kwargs*)

circuits.web.main.**main**()

circuits.web.main.**parse_bind**(*bind*)

circuits.web.main.**parse_options**()

circuits.web.main.**select_poller**(*poller*)

### 2.5.58 circuits.web.parsers package

#### Submodules

##### circuits.web.parsers.http module

**class** circuits.web.parsers.http.**HttpParser**(*kind=2*, *decompress=False*)
Bases: object

**execute**(*data*, *length*)

**get_headers**()

**get_method**()

**get_path**()

**get_query_string**()

**get_scheme**()

**get_status_code**()

**get_url**()

**get_version**()

**is_chunked**()
return True if Transfer-Encoding header value is chunked

**is_headers_complete**()
return True if all headers have been parsed.

**is_message_begin**()
return True if the parsing start

**is_message_complete**()
return True if the parsing is done (we get EOF)

**is_partial_body**()
return True if a chunk of body have been parsed

**is_upgrade**()
Do we get upgrade header in the request. Useful for websockets

**recv_body**()
return last chunk of the parsed body

**recv_body_into**(*barray*)
Receive the last chunk of the parsed bodyand store the data in a buffer rather than creating a new string.

**should_keep_alive**()
return True if the connection should be kept alive

**exception** circuits.web.parsers.http.**InvalidChunkSize**
Bases: exceptions.Exception

error raised when we parse an invalid chunk size

**exception** circuits.web.parsers.http.**InvalidHeader**
Bases: exceptions.Exception

error raised on invalid header

**exception** circuits.web.parsers.http.**InvalidRequestLine**
Bases: exceptions.Exception

error raised when first line is invalid

### circuits.web.parsers.multipart module

**Parser for multipart/form-data** This module provides a parser for the multipart/form-data format. It can read from a file, a socket or a WSGI environment. The parser can be used to replace cgi.FieldStorage (without the bugs) and works with Python 2.5+ and 3.x (2to3).

**Licence (MIT)**

Copyright (c) 2010, Marcel Hellkamp. Inspired by the Werkzeug library: http://werkzeug.pocoo.org/

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**class** circuits.web.parsers.multipart.**MultiDict**(**a*, ***k*)
Bases: _abcoll.MutableMapping

A dict that remembers old values for each key

**append**(*key*, *value*)

**get**(*key*, *default=None*, *index=-1*)

**getall**(*key*)

**iterallitems**()

**keys**()

**replace**(*key*, *value*)

**exception** circuits.web.parsers.multipart.**MultipartError**
Bases: exceptions.ValueError

**class** circuits.web.parsers.multipart.**MultipartParser**(*stream*, *boundary*, *content_length=-1*, *disk_limit=1073741824*, *mem_limit=1048576*, *memfile_limit=262144*, *buffer_size=65536*, *charset='latin1'*)
Bases: object

Parse a multipart/form-data byte stream. This object is an iterator over the parts of the message.

**Parameters**

- **stream** – A file-like stream. Must implement `.read(size)`.
- **boundary** – The multipart boundary as a byte string.
- **content_length** – The maximum number of bytes to read.

**get**(*name*, *default=None*)
Return the first part with that name or a default value (None).

**get_all**(*name*)
Return a list of parts with that name.

**parts**()
Returns a list with all parts of the multipart message.

**class** circuits.web.parsers.multipart.**MultipartPart**(*buffer_size=65536*, *memfile_limit=262144*, *charset='latin1'*)

Bases: object

**feed**(*line*, *nl=''*)

**finish_header**()

**is_buffered**()
Return true if the data is fully buffered in memory.

**save_as**(*path*)

**value**
Data decoded with the specified charset

**write_body**(*line*, *nl*)

**write_header**(*line*, *nl*)

circuits.web.parsers.multipart.**copy_file**(*stream*, *target*, *maxread=-1*, *buffer_size=32*)
Read from :stream and write to :target until :maxread or EOF.

circuits.web.parsers.multipart.**header_quote**(*val*)

circuits.web.parsers.multipart.**header_unquote**(*val*, *filename=False*)

circuits.web.parsers.multipart.**parse_form_data**(*environ*, *charset='utf8'*, *strict=False*, ***kw*)
Parse form data from an environ dict and return a (forms, files) tuple. Both tuple values are dictionaries with the form-field name as a key (text_type) and lists as values (multiple values per key are possible). The forms-dictionary contains form-field values as text_type strings. The files-dictionary contains *MultipartPart* instances, either because the form-field was a file-upload or the value is to big to fit into memory limits.

**Parameters**

- **environ** – An WSGI environment dict.
- **charset** – The charset to use if unsure. (default: utf8)
- **strict** – If True, raise *MultipartError* on any parsing errors. These are silently ignored by default.

circuits.web.parsers.multipart.**parse_options_header**(*header*, *options=None*)

circuits.web.parsers.multipart.**tob**(*data*, *enc='utf8'*)

#### circuits.web.parsers.querystring module

**class** circuits.web.parsers.querystring.**QueryStringParser**(*data*)
Bases: object

**parse**(*key*, *value*)

**process**(*pair*)

**tokens**(*key*)

**class** circuits.web.parsers.querystring.**QueryStringToken**
Bases: object

**ARRAY** = 'ARRAY'

**KEY** = 'KEY'

**OBJECT** = 'OBJECT'

### Module contents

circuits.web parsers

## 2.5.59 circuits.web.parsers.http module

**class** circuits.web.parsers.http.**HttpParser**(*kind=2*, *decompress=False*)
Bases: object

**execute**(*data*, *length*)

**get_headers**()

**get_method**()

**get_path**()

**get_query_string**()

**get_scheme**()

**get_status_code**()

**get_url**()

**get_version**()

**is_chunked**()
return True if Transfer-Encoding header value is chunked

**is_headers_complete**()
return True if all headers have been parsed.

**is_message_begin**()
return True if the parsing start

**is_message_complete**()
return True if the parsing is done (we get EOF)

**is_partial_body**()
return True if a chunk of body have been parsed

**is_upgrade**()
Do we get upgrade header in the request. Useful for websockets

**recv_body**()
return last chunk of the parsed body

**recv_body_into**(*barray*)
Receive the last chunk of the parsed bodyand store the data in a buffer rather than creating a new string.

**should_keep_alive**()
return True if the connection should be kept alive

**exception** circuits.web.parsers.http.**InvalidChunkSize**
Bases: exceptions.Exception

error raised when we parse an invalid chunk size

**exception** circuits.web.parsers.http.**InvalidHeader**
Bases: exceptions.Exception

error raised on invalid header

**exception** circuits.web.parsers.http.**InvalidRequestLine**
Bases: exceptions.Exception

error raised when first line is invalid

### 2.5.60 circuits.web.parsers.multipart module

#### Parser for multipart/form-data

This module provides a parser for the multipart/form-data format. It can read from a file, a socket or a WSGI environment. The parser can be used to replace cgi.FieldStorage (without the bugs) and works with Python 2.5+ and 3.x (2to3).

##### Licence (MIT)

Copyright (c) 2010, Marcel Hellkamp. Inspired by the Werkzeug library: http://werkzeug.pocoo.org/

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

**class** circuits.web.parsers.multipart.**MultiDict**(**a*, ***k*)
Bases: _abcoll.MutableMapping

A dict that remembers old values for each key

**append**(*key*, *value*)

**get**(*key*, *default=None*, *index=-1*)

**getall**(*key*)

**iterallitems**()

**keys**()

**replace**(*key*, *value*)

**exception** circuits.web.parsers.multipart.**MultipartError**
Bases: exceptions.ValueError

**class** circuits.web.parsers.multipart.**MultipartParser**(*stream*, *boundary*, *content_length=-1*, *disk_limit=1073741824*, *mem_limit=1048576*, *memfile_limit=262144*, *buffer_size=65536*, *charset='latin1'*)
Bases: object

Parse a multipart/form-data byte stream. This object is an iterator over the parts of the message.

**Parameters**

- **stream** – A file-like stream. Must implement .read(size).
- **boundary** – The multipart boundary as a byte string.
- **content_length** – The maximum number of bytes to read.

**get**(*name*, *default=None*)
Return the first part with that name or a default value (None).

**get_all**(*name*)
Return a list of parts with that name.

**parts**()
Returns a list with all parts of the multipart message.

**class** circuits.web.parsers.multipart.**MultipartPart**(*buffer_size=65536*, *memfile_limit=262144*, *charset='latin1'*)
Bases: object

**feed**(*line*, *nl=''*)

**finish_header**()

**is_buffered**()
Return true if the data is fully buffered in memory.

**save_as**(*path*)

**value**
Data decoded with the specified charset

**write_body**(*line*, *nl*)

**write_header**(*line*, *nl*)

circuits.web.parsers.multipart.**copy_file**(*stream*, *target*, *maxread=-1*, *buffer_size=32*)
Read from :stream and write to :target until :maxread or EOF.

circuits.web.parsers.multipart.**header_quote**(*val*)

circuits.web.parsers.multipart.**header_unquote**(*val*, *filename=False*)

circuits.web.parsers.multipart.**parse_form_data**(*environ*, *charset='utf8'*, *strict=False*, ***kw*)

Parse form data from an environ dict and return a (forms, files) tuple. Both tuple values are dictionaries with the form-field name as a key (text_type) and lists as values (multiple values per key are possible). The forms-dictionary contains form-field values as text_type strings. The files-dictionary contains *MultipartPart* instances, either because the form-field was a file-upload or the value is to big to fit into memory limits.

**Parameters**

- **environ** – An WSGI environment dict.
- **charset** – The charset to use if unsure. (default: utf8)
- **strict** – If True, raise *MultipartError* on any parsing errors. These are silently ignored by default.

circuits.web.parsers.multipart.**parse_options_header**(*header*, *options=None*)

circuits.web.parsers.multipart.**tob**(*data*, *enc='utf8'*)

### 2.5.61 circuits.web.parsers.querystring module

**class** circuits.web.parsers.querystring.**QueryStringParser**(*data*)

Bases: object

**parse**(*key*, *value*)

**process**(*pair*)

**tokens**(*key*)

**class** circuits.web.parsers.querystring.**QueryStringToken**

Bases: object

**ARRAY** = 'ARRAY'

**KEY** = 'KEY'

**OBJECT** = 'OBJECT'

### 2.5.62 circuits.web.processors module

circuits.web.processors.**process**(*request*, *params*)

circuits.web.processors.**process_multipart**(*request*, *params*)

circuits.web.processors.**process_urlencoded**(*request*, *params*, *encoding='utf-8'*)

### 2.5.63 circuits.web.servers module

Web Servers

This module implements the several Web Server components.

**class** circuits.web.servers.**BaseServer**(*bind*, *encoding='utf-8'*, *secure=False*, *certfile=None*, *channel='web'*, *display_banner=True*)

Bases: *circuits.core.components.BaseComponent*

Create a Base Web Server

Create a Base Web Server (HTTP) bound to the IP Address / Port or UNIX Socket specified by the 'bind' parameter.

**Variables** ***server*** – Reference to underlying Server Component

**Parameters** **bind** (*Instance of int, list, tuple or str*) – IP Address / Port or UNIX Socket to bind to.

The 'bind' parameter is quite flexible with what valid values it accepts.

If an int is passed, a TCPServer will be created. The Server will be bound to the Port given by the 'bind' argument and the bound interface will default (normally to "0.0.0.0").

If a list or tuple is passed, a TCPServer will be created. The Server will be bound to the Port given by the 2nd item in the 'bind' argument and the bound interface will be the 1st item.

If a str is passed and it contains the ':' character, this is assumed to be a request to bind to an IP Address / Port. A TCpServer will thus be created and the IP Address and Port will be determined by splitting the string given by the 'bind' argument.

Otherwise if a str is passed and it does not contain the ':' character, a file path is assumed and a UNIXServer is created and bound to the file given by the 'bind' argument.

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**channel** = 'web'

**display_banner**

**host**

**port**

**secure**

**class** circuits.web.servers.**FakeSock**

**getpeername**()

**class** circuits.web.servers.**Server**(*bind*, ***kwargs*)

Bases: *circuits.web.servers.BaseServer*

Create a Web Server

Create a Web Server (HTTP) complete with the default Dispatcher to parse requests and posted form data dispatching to appropriate Controller(s).

See: circuits.web.servers.BaseServer

x.__init__(...) initializes x; see x.__class__.__doc__ for signature

**class** circuits.web.servers.**StdinServer**(*encoding='utf-8'*, *channel='web'*)

Bases: *circuits.core.components.BaseComponent*

**channel** = 'web'

**host**

**port**

**read**(*data*)

**secure**

**write**(*sock*, *data*)

### 2.5.64 circuits.web.sessions module

Session Components

This module implements Session Components that can be used to store and access persistent information.

**class** circuits.web.sessions.**Sessions**(*name='circuits.session'*, *channel='web'*)
Bases: *circuits.core.components.Component*

**channel = 'web'**

**load**(*sid*)

**request**(*request*, *response*)

**save**(*sid*, *data*)
Save User Session Data for sid

circuits.web.sessions.**create_session**(*request*)
Create a unique session id from the request

Returns a unique session using `uuid4()` and a `sha1()` hash of the users IP Address and User Agent in the form of `sid/who`.

circuits.web.sessions.**verify_session**(*request*, *sid*)
Verify a User's Session

This verifies the User's Session by verifying the SHA1 Hash of the User's IP Address and User-Agent match the provided Session ID.

circuits.web.sessions.**who**(*request*, *encoding='utf-8'*)
Create a SHA1 Hash of the User's IP Address and User-Agent

### 2.5.65 circuits.web.tools module

Tools

This module implements tools used throughout circuits.web. These tools can also be used within Controlelrs and request handlers.

**class** circuits.web.tools.**ReverseProxy**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**headers = ('X-Real-IP', 'X-Forwarded-For')**

**init**(*headers=None*)
Web Component for identifying the original client IP when a reverse proxy is used

**Parameters** **headers** – List of HTTP headers to read the original client IP

circuits.web.tools.**basic_auth**(*request*, *response*, *realm*, *users*, *encrypt=None*)
Perform Basic Authentication

If auth fails, returns an Unauthorized error with a basic authentication header.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.
- **encrypt** (*callable*) – Callable used to encrypt the password returned from the user-agent. if None it defaults to a md5 encryption.

circuits.web.tools.**check_auth**(*request*, *response*, *realm*, *users*, *encrypt=None*)
Check Authentication

If an Authorization header contains credentials, return True, else False.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.
- **encrypt** (*callable*) – Callable used to encrypt the password returned from the user-agent. if None it defaults to a md5 encryption.

circuits.web.tools.**digest_auth**(*request*, *response*, *realm*, *users*)
Perform Digest Authentication

If auth fails, raise 401 with a digest authentication header.

**Parameters**

- **realm** (*str*) – The authentication realm.
- **users** (*dict or callable*) – A dict of the form: {username: password} or a callable returning a dict.

circuits.web.tools.**expires**(*request*, *response*, *secs=0*, *force=False*)
Tool for influencing cache mechanisms using the 'Expires' header.

'secs' must be either an int or a datetime.timedelta, and indicates the number of seconds between response.time and when the response should expire. The 'Expires' header will be set to (response.time + secs).

If 'secs' is zero, the 'Expires' header is set one year in the past, and the following "cache prevention" headers are also set: - 'Pragma': 'no-cache' - 'Cache-Control': 'no-cache, must-revalidate'

If 'force' is False (the default), the following headers are checked: 'Etag', 'Last-Modified', 'Age', 'Expires'. If any are already present, none of the above response headers are set.

circuits.web.tools.**gzip**(*response*, *level=4*, *mime_types=('text/html', 'text/plain')*)
Try to gzip the response body if Content-Type in mime_types.

response.headers['Content-Type'] must be set to one of the values in the mime_types arg before calling this function.

**No compression is performed if any of the following hold:**

- The client sends no Accept-Encoding request header
- No 'gzip' or 'x-gzip' is present in the Accept-Encoding header
- No 'gzip' or 'x-gzip' with a qvalue > 0 is present
- The 'identity' value is given with a qvalue > 0.

circuits.web.tools.**serve_download**(*request*, *response*, *path*, *name=None*)
Serve 'path' as an application/x-download attachment.

circuits.web.tools.**serve_file**(*request*, *response*, *path*, *type=None*, *disposition=None*, *name=None*)
Set status, headers, and body in order to serve the given file.

The Content-Type header will be set to the type arg, if provided. If not provided, the Content-Type will be guessed by the file extension of the 'path' argument.

If disposition is not None, the Content-Disposition header will be set to "<disposition>; filename=<name>". If name is None, it will be set to the basename of path. If disposition is None, no Content-Disposition header will be written.

circuits.web.tools.**validate_etags**(*request*, *response*, *autotags=False*)
Validate the current ETag against If-Match, If-None-Match headers.

If autotags is True, an ETag response-header value will be provided from an MD5 hash of the response body (unless some other code has already provided an ETag header). If False (the default), the ETag will not be automatic.

WARNING: the autotags feature is not designed for URL's which allow methods other than GET. For example, if a POST to the same URL returns no content, the automatic ETag will be incorrect, breaking a fundamental use for entity tags in a possibly destructive fashion. Likewise, if you raise 304 Not Modified, the response body will be empty, the ETag hash will be incorrect, and your application will break. See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.24

circuits.web.tools.**validate_since**(*request*, *response*)
Validate the current Last-Modified against If-Modified-Since headers.

If no code has set the Last-Modified response header, then no validation will be performed.

### 2.5.66 circuits.web.url module

This is a module for dealing with urls. In particular, sanitizing them.

**class** circuits.web.url.**URL**(*scheme*, *host*, *port*, *path*, *params=''*, *query=''*, *fragment=''*)
Bases: object

**For more information on how and what we parse / sanitize:** http://tools.ietf.org/html/rfc1808.html

**The more up-to-date RFC is this one:** http://www.ietf.org/rfc/rfc3986.txt

**absolute**()
Return True if this is a fully-qualified URL with a hostname and everything

**abspath**()
Clear out any '..' and excessive slashes from the path

**canonical**()
Canonicalize this url. This includes reordering parameters and args to have a consistent ordering

**defrag**()
Remove the fragment from this url

**deparam**(*params=None*)
Strip any of the provided parameters out of the url

**encode**(*encoding*)
Return the url in an arbitrary encoding

**equiv**(*other*)
Return true if this url is equivalent to another

**escape**()
Make sure that the path is correctly escaped

**lower**()
Lowercase the hostname

**classmethod parse**(*url*, *encoding*)
Parse the provided url, and return a URL instance

**punycode**()
Convert to punycode hostname

**relative**(*path*, *encoding='utf-8'*)
Evaluate the new path relative to the current url

**sanitize**()
A shortcut to abspath, escape and lowercase

**unescape**()
Unescape the path

**unicode**()
Return a unicode version of this url

**unpunycode**()
Convert to an unpunycoded hostname

**utf8**()
Return a utf-8 version of this url

circuits.web.url.**parse_url**(*url*, *encoding='utf-8'*)
Parse the provided url string and return an URL object

### 2.5.67 circuits.web.utils module

Utilities

This module implements utility functions.

circuits.web.utils.**average**(*xs*)

circuits.web.utils.**compress**(*body*, *compress_level*)
Compress 'body' at the given compress_level.

circuits.web.utils.**dictform**(*form*)

circuits.web.utils.**get_ranges**(*headervalue*, *content_length*)
Return a list of (start, stop) indices from a Range header, or None.

Each (start, stop) tuple will be composed of two ints, which are suitable for use in a slicing operation. That is, the header "Range: bytes=3-6", if applied against a Python string, is requesting resource[3:7]. This function will return the list [(3, 7)].

If this function returns an empty list, you should return HTTP 416.

circuits.web.utils.**parse_body**(*request*, *response*, *params*)

circuits.web.utils.**parse_qs**(*query_string*) → dict
Build a params dictionary from a query_string. If keep_blank_values is True (the default), keep values that are blank.

circuits.web.utils.**stddev**(*xs*)

circuits.web.utils.**variance**(*xs*)

## 2.5.68 circuits.web.websockets package

### Submodules

#### circuits.web.websockets.client module

**class** circuits.web.websockets.client.**WebSocketClient**(*url*, *channel='wsclient'*, *wschannel='ws'*, *headers=None*)

Bases: *circuits.core.components.BaseComponent*

An RFC 6455 compliant WebSocket client component. Upon receiving a `circuits.web.client.Connect` event, the component tries to establish the connection to the server in a two stage process. First, a *circuits.net.events.connect* event is sent to a child *TCPClient*. When the TCP connection has been established, the HTTP request for opening the WebSocket is sent to the server. A failure in this setup process is signaled by raising an *NotConnected* exception.

When the server accepts the request, the WebSocket connection is established and can be used very much like an ordinary socket by handling *read* events on and sending *write* events to the channel specified as the `wschannel` parameter of the constructor. Firing a *close* event on that channel closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

**Parameters**

- **url** – the URL to connect to.
- **channel** – the channel used by this component
- **wschannel** – the channel used for the actual WebSocket communication (read, write, close events)
- **headers** – additional headers to be passed with the WebSocket setup HTTP request

**channel = 'wsclient'**

**close**()

**connected**

#### circuits.web.websockets.dispatcher module

**class** circuits.web.websockets.dispatcher.**WebSocketsDispatcher**(*path=None*, *wschannel='wsserver'*, **args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

This class implements an RFC 6455 compliant WebSockets dispatcher that handles the WebSockets handshake and upgrades the connection.

The dispatcher listens on its channel for `Request` events and tries to match them with a given path. Upon a match, the request is checked for the proper Opening Handshake information. If successful, the dispatcher confirms the establishment of the connection to the client. Any subsequent data from the client is handled as a WebSocket data frame, decoded and fired as a `Read` event on the `wschannel` passed to the constructor. The data from *write* events on that channel is encoded as data frames and forwarded to the client.

Firing a `Close` event on the `wschannel` closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

**Parameters**

- **path** – the path to handle. Requests that start with this path are considered to be WebSocket Opening Handshakes.
- **wschannel** – the channel on which `read` events from the client will be delivered and where *write* events to the client will be sent to.

**channel = 'web'**

### Module contents

circuits.web websockets

## 2.5.69 circuits.web.websockets.client module

**class** circuits.web.websockets.client.**WebSocketClient**(*url*, *channel='wsclient'*, *wschannel='ws'*, *headers=None*)

Bases: *circuits.core.components.BaseComponent*

An RFC 6455 compliant WebSocket client component. Upon receiving a `circuits.web.client.Connect` event, the component tries to establish the connection to the server in a two stage process. First, a *circuits.net.events.connect* event is sent to a child *TCPClient*. When the TCP connection has been established, the HTTP request for opening the WebSocket is sent to the server. A failure in this setup process is signaled by raising an *NotConnected* exception.

When the server accepts the request, the WebSocket connection is established and can be used very much like an ordinary socket by handling *read* events on and sending *write* events to the channel specified as the `wschannel` parameter of the constructor. Firing a *close* event on that channel closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

**Parameters**

- **url** – the URL to connect to.
- **channel** – the channel used by this component
- **wschannel** – the channel used for the actual WebSocket communication (read, write, close events)
- **headers** – additional headers to be passed with the WebSocket setup HTTP request

**channel = 'wsclient'**

**close**()

**connected**

## 2.5.70 circuits.web.websockets.dispatcher module

**class** circuits.web.websockets.dispatcher.**WebSocketsDispatcher**(*path=None*, *wschannel='wsserver'*, **args*, ***kwargs*)

Bases: *circuits.core.components.BaseComponent*

This class implements an RFC 6455 compliant WebSockets dispatcher that handles the WebSockets handshake and upgrades the connection.

The dispatcher listens on its channel for `Request` events and tries to match them with a given path. Upon a match, the request is checked for the proper Opening Handshake information. If successful, the dispatcher confirms the establishment of the connection to the client. Any subsequent data from the client is handled as a

WebSocket data frame, decoded and fired as a `Read` event on the `wschannel` passed to the constructor. The data from *`write`* events on that channel is encoded as data frames and forwarded to the client.

Firing a `Close` event on the `wschannel` closes the connection in an orderly fashion (i.e. as specified by the WebSocket protocol).

**Parameters**

- **`path`** – the path to handle. Requests that start with this path are considered to be WebSocket Opening Handshakes.
- **`wschannel`** – the channel on which `read` events from the client will be delivered and where *`write`* events to the client will be sent to.

**`channel` = 'web'**

### 2.5.71 circuits.web.wrappers module

Request/Response Wrappers

This module implements the Request and Response objects.

**class** `circuits.web.wrappers.`**`Body`**
Bases: `object`

Response Body

**class** `circuits.web.wrappers.`**`HTTPStatus`**(*status=200*, *reason=None*)
Bases: `object`

**`reason`**

**`status`**

**class** `circuits.web.wrappers.`**`Host`**(*ip*, *port*, *name=None*)
Bases: `object`

An internet address.

name should be the client's host name. If not available (because no DNS lookup is performed), the IP address should be used instead.

**`ip` = '0.0.0.0'**

**`name` = 'unknown.tld'**

**`port` = 80**

**class** `circuits.web.wrappers.`**`Request`**(*sock*, *method='GET'*, *scheme='http'*, *path='/'*, *protocol=(1, 1)*, *qs=''*, *headers=None*, *server=None*)
Bases: `object`

Creates a new Request object to hold information about a request.

**Parameters**

- **`sock`** (*socket.socket*) – The socket object of the request.
- **`method`** (*str*) – The requested method.
- **`scheme`** (*str*) – The requested scheme.
- **`path`** (*str*) – The requested path.
- **`protocol`** (*str*) – The requested protocol.

- **qs** (*str*) – The query string of the request.

initializes x; see x.__class__.__doc__ for signature

**handled = False**

**host = ‘’**

**index = None**

**local = Host(‘127.0.0.1’, 80, ‘127.0.0.1’)**

**login = None**

**protocol = (1, 1)**

**remote = Host(‘’, 0, ‘’)**

**scheme = ‘http’**

**script_name = ‘’**

**server = None**

**class** circuits.web.wrappers.**Response**(*sock*, *request*) → new Response object
Bases: object

A Response object that holds the response to send back to the client. This ensure that the correct data is sent in the correct order.

initializes x; see x.__class__.__doc__ for signature

**body**
Response Body

**chunked = False**

**close = False**

**done = False**

**prepare**()

**status**
Response Status

**stream = False**

**class** circuits.web.wrappers.**Status**
Bases: object

Response Status

circuits.web.wrappers.**file_generator**(*input*, *chunkSize=4096*)

### 2.5.72 circuits.web.wsgi module

WSGI Components

This module implements WSGI Components.

**class** circuits.web.wsgi.**Application**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = ‘web’**

**getRequestResponse**(*environ*)

**headerNames = {'CONTENT_LENGTH': 'Content-Length', 'REMOTE_HOST': 'Remote-Host', 'CONTENT_TYPE':**

**host**

**init**()

**port**

**response**(*event*, *response*)

**secure**

**translateHeaders**(*environ*)

**class** circuits.web.wsgi.**Gateway**(**args*, ***kwargs*)
Bases: *circuits.core.components.BaseComponent*

initializes x; see x.__class__.__doc__ for signature

**channel = 'web'**

**init**(*apps*)

circuits.web.wsgi.**create_environ**(*errors*, *path*, *req*)

## 2.6 Developer Docs

So, you'd like to contribute to circuits in some way? Got a bug report? Having problems running the examples? Having problems getting circuits working in your environment?

Excellent. Here's what you need to know.

### 2.6.1 Development Introduction

Here's how we do things in circuits...

#### Communication

- IRC Channel on the FreeNode IRC Network
- Developer Mailing List
- Issue Tracker

---

**Note:** If you are familiar with IRC and use your own IRC Client then connect to the FreeNode Network and `/join #circuits`.

---

#### Standards

We use the following coding standard:

- PEP-008

We also lint our codebase with the following tools:

- pyflakes

- pep8
- mccabe

Please ensure your Development IDE or Editor has the above linters and checkers in place and enabled.

Alternatively you can use the following command line tool:

- flake8

#### Tools

We use the following tools to develop circuits and share code:

- **Code Sharing:** Git
- **Code Hosting and Bug Reporting:** GitHub
- **Issue Tracker:** Issue Tracker
- **Documentation Hosting:** Read the Docs
- **Package Hosting:** Python Package Index (PyPi)
- **Continuous Integration:** Travis CI
- **Code Quality:** Landscape

### 2.6.2 Contributing to circuits

Here's how you can contribute to circuits

#### Share your story

One of the best ways you can contribute to circuits is by using circuits. Share with us your story of how you've used circuits to solve a problem or create a new software solution using the circuits framework and library of components.

#### Submitting Bug Reports

We welcome all bug reports. We do however prefer bug reports in a clear and concise form with repeatable steps. One of the best ways you can report a bug to us is by writing a unit test (//similar to the ones in our tests//) so that we can verify the bug, fix it and commit the fix along with the test.

To submit a bug report, please Create an Issue

#### Writing new tests

We're not perfect, and we're still writing more tests to ensure quality code. If you'd like to help, please Fork circuits, write more tests that cover more of our code base and submit a Pull Request. Many Thanks!

#### Adding New Features

If you'd like to see a new feature added to circuits, then we'd like to hear about it~ We would like to see some discussion around any new features as well as valid use-cases. To start the discussions off, please either:

- Chat to us on #circuits on the FreeNode IRC Network

  or

- Create an Issue

### 2.6.3 Setting up a circuits Development Environment

This is the recommended way to setup a development enviornment for developing with or on circuits.

---

**Note:** This document *assumes* you already have a working Python environment with a minimum Python version of 2.6 as well as pip already installed.

---

#### Prequisites

It is highly recommended that you install and use virtualenv for all your Python development and production deployments (*not just circuits*).

It is also convenient to install and use the accompanying shell scripts and tools virtualenvwrapper which adds a nice set of workflows and functions useful for both development and deployments.

```
$ pip install -U virtualenvwrapper
$ source $(which virtualenvwrapper.sh)
```

---

**Note:** You should put `source $(which virtualenvwrapper.sh)` in either your `$HOME/.bashrc` or `$HOME/.profile` depending on how you login and interact with your terminals.

---

In addition to the above recommendations you must also have a Git client installed and ready to use as well as your Editor/IDE of choice ready to use.

#### Getting Started

1. Fork circuits (*if you haven't done so already*)
2. Clone your forked repository using Git
3. Create a new virtual environment using virtualenvwrapper
4. Install the Development Requirements
5. Install circuits in "develop" mode

And you're done!

Example:

```
$ git clone git@github.com:prologic/circuits.git
$ cd circuits
$ mkvirtualenv circuits
$ pip install -r requirements-dev.txt
$ python setup.py develop
```

Alternatively if you already have Fabric installed:

```
$ git clone git@github.com:prologic/circuits.git
$ cd circuits
$ mkvirtualenv circuits
$ fab develop
```

### 2.6.4 Development Processes

We document all our internal development processes here so you know exactly how we work and what to expect. If you find any issues or problems please let us know!

#### Software Development Life Cycle (SDLC)

We employ the use of the SCRUM Agile Process and use our Issue Tracker to track features, bugs, chores and releases. If you wish to contribute to circuits, please familiarize yourself with SCRUM and GitHub's Issue Tracker.

#### Bug Reports

- New Bug Reports are submitted via: https://github.com/circuits/circuits/issues
- Confirmation and Discussion of all New Bug Reports.
- Once confirmed, a new Bug is raised in our Issue Tracker
- An appropriate milestone will be set (*depending on current milestone's schedule and resources*)
- A unit test developed that demonstrates the bug's failure.
- A fix developed that passes the unit test and breaks no others.
- A New Pull Request created with the fix.

  This must contains: - A new or modified unit test. - A patch that fixes the bug ensuring all unit tests pass. - The Change Log updated. - Appropriate documentation updated.
- The Pull Request is reviewed and approved by at least two other developers.

#### Feature Requests

- New Feature Requests are submitted via: https://github.com/circuits/circuits/issues
- Confirmation and Discussion of all New Feature Requests.
- Once confirmed, a new Feature is raised in our Issue Tracker
- An appropriate milestone will be set (*depending on current milestone's schedule and resources*)
- A unit test developed that demonstrates the new feature.
- The new feature developed that passes the unit test and breaks no others.
- A New Pull Request created with the fix.

  This must contains: - A new or modified unit test. - A patch that implements the new feature ensuring all unit tests pass. - The Change Log updated. - Appropriate documentation updated.
- The Pull Request is reviewed and approved by at least two other developers.

#### Writing new Code

- Submit a New Issue
- Write your code.
- Use flake8 to ensure code quality.
- Run the tests:

```
$ tox
```

- Ensure any new or modified code does not break existing unit tests.
- Update any relevant doc strings or documentation.
- Update the Change Log updated.
- Submit a New Pull Request.

#### Running the Tests

To run the tests you will need the following installed:

- tox installed as well as
- pytest-cov
- pytest

All of these can be installed via `pip`.

Please also ensure that you you have all supported versions of Python that circuits supports installed in your local environment.

To run the tests:

```
$ tox
```

### 2.6.5 Development Standards

We use the following development standards:

#### Cyclomatic Complexity

- Code Complexity shall not exceed `10`

  See: Limiting Cyclomatic Complexity

#### Coding Style

- Code shall confirm to the PEP8 Style Guide.

**Note:** This includes the 79 character limit!

- Doc Strings shall confirm to the PEP257 Convention.

**Note:** Arguments, Keyword Arguments, Return and Exceptions must be documented with the appropriate Sphinx'Python Domain <http://sphinx-doc.org/latest/domains.html#the-python-domain>'_.

### Revision History

- Commits shall be small tangible pieces of work. - Each commit must be concise and manageable. - Large changes are to be done over smaller commits.
- There shall be no commit squashing.
- Rebase your changes as often as you can.

### Unit Tests

- Every new feature and bug fix must be accompanied with a unit test. (*The only exception to this are minor trivial changes*).

## 2.7 Change Log

- #119: Fixed bug in `circuits.web.url.parse_url()` that caused a display issue with port bindings on ports 80 and 443.
- #115: Fixed FallbackErrorHandler API Change
- #113: Fixed bug with forced shutdown of subprocesses in Windows.
- : Bridge: Send exceptions via brige before change the exceptions weren't propagated via bridge because traceback object is not pickable, now traceback object is replaced by corresponding traceback list
- : Bridge: Do not propagate no results via bridge
- : Fixed issue in brige with ommiting all but the first events sent at once
- : Fixed exception handing in circuits.web
- : Fixed import of FallBackExceptionHandler
- : Node: Add node examples.
- : Node: fixes event response flood.
- : Node: fixes the event value issue.
- : Node: add event firewall (client / server).
- : Node: add peer node: return channel name.
- : Fixes optional parameters handling (client / server).
- : Rename the FallbackErrorHandler to FallbackExceptionHandler and the event it listens to to exception
- : Bridge waits for event processing on the other side before proxy handler ends. Now it is possible to collect values from remote handlers in %_success event.
- #96: Link to ChangeLog from README
- #117: Fixed inconsistent top-level examples.
- #94: Modified the `circuits.web.Logger` to use the `response_success` event.

- #98: Dockerized circuits. See: https://docker.io/
- #99: Added Digest Auth support to the `circuits.web` CLI Tool
- #103: Added the firing of a `disconnect` event for the WebSocketsDispatcher.
- #108: Improved server support for the IRC Protocol.
- #112: Improved Signal Handling
- #37: Fixed a typo in *`File`*
- #38: Guard against invalid headers. (circuits.web)
- #46: Set `Content-Type` header on response for errors. (circuits.web)
- #48: Allow `event` to be passed to the decorated function (*the request handler*) for circuits.web
- #45: Fixed use of `cmp()` and `__cmp__()` for Python 3 compatibility.
- #56: circuits.web HEAD request send response body web
- #62: Fix packaging and bump circuits 1.5.1 for @dsuch (*Dariusz Suchojad*) for Zato
- #53: WebSocketClient treating WebSocket data in same TCP segment as HTTP response as part the HTTP response. web
- #67: web example jsontool is broken on python3 web
- #77: Uncaught exceptions Event collides with sockets and others core
- #81: "index" method not serving / web
- #76: Missing unit test for DNS lookup failures net
- #66: web examples jsonserializer broken web
- #59: circuits.web DoS in serve_file (remote denial of service) web
- #91: Call/Wait and specific instances of events
- #89: Class attribtues that reference methods cause duplicate event handlers core
- #47: Dispatcher does not fully respect optional arguments. web
- #97: Fixed `tests.net.test_tcp.test_lookup_failure` test for Windows
- #100: Fixed returned Content-Type in JSON-RPC Dispatcher.
- #102: Fixed minor bug with WebSocketsDispatcher causing superflusous `connect()` events from being fired.
- #104: Prevent other websockets sessions from closing.
- #106: Added `__format__` method to circuits.web.wrappers.HTTPStatus.
- #107: Added `__le__` and `__ge__` methods to `circuits.web.wrappers.HTTPStatus`
- #109: Fixed `Event.create()` factory and metaclass.
- #111: Fixed broken Digest Auth Test for circuits.web
- #63: typos in documentation docs
- #60: meantion @handler decorator in tutorial docs
- #65: Update tutorial to match circuits 3.0 API(s) and Semantics docs
- #69: Merge #circuits-dev FreeNode Channel into #circuits
- #75: Document and show examples of using circuits.tools docs

- #70: Convention around method names of event handlers
- #72: Update Event Filtering section of Users Manual docs
- #73: Fix duplication in auto generated API Docs. docs
- #78: Migrate Change Log maintenance and build to Releases
- #71: Document the value_changed event docs
- #92: Update circuitsframework.com content docs
- #88: Document the implicit registration of components attached as class attributes docs
- #87: A rendered example of `circuits.tools.graph()`. docs
- #85: Migrate away from ShiningPanda
- #61: circuits.web documentation enhancements docs
- #86: Telnet Tutorial
- #95: Updated Developer Documentation with corrections and a new workflow.

### 2.7.1 Older Change Logs

For older Change Logs of previous versions of circuits please see the respective PyPi page(s):

- circuits-2.1.0
- circuits-2.0.1
- circuits-2.0.0
- circuits-1.6
- circuits-1.5

## 2.8 Road Map

We managed our Roadmap on our [Github Project](https://github.com/circuits/circuits)

## 2.9 Contributors

circuits was originally designed, written and primarily maintained by James Mills (http://prologic.shortcircuit.net.au/).

The following users and developers have contributed to circuits:

- Alessio Deiana
- Dariusz Suchojad
- Tim Miller
- Holger Krekel
- Justin Giorgi
- Edwin Marshall
- Alex Mayfield

- Toni Alatalo
- Michael Lipp
- Matthieu Chevrier
- Yoann Ono Dit Biot

Anyone not listed here (*apologies as this list is taken directly from Mercurial's churn command and output*). We appreciate any and all contributions to circuits.

## 2.10 Frequently Asked Questions

### 2.10.1 General

**... What is circuits?** circuits is an event-driven framework with a high focus on Component architectures making your life as a software developer much easier. circuits allows you to write maintainable and scalable systems easily

**... Can I write networking applications with circuits?** Yes absolutely. circuits comes with socket I/O components for tcp, udp and unix sockets with asynchronous polling implementations for select, poll, epoll and kqueue.

**... Can I integrate circuits with a GUI library?** This is entirely possible. You will have to hook into the GUI's main loop.

**... What are the core concepts in circuits?** Components and Events. Components are maintainable reusable units of behavior that communicate with other components via a powerful message passing system.

**... How would you compare circuits to Twisted?** Others have said that circuits is very elegant in terms of it's usage. circuits' component architecture allows you to define clear interfaces between components while maintaining a high level of scalability and maintainability.

**... Can Components communicate with other processes?** Yes. circuits implements currently component bridging and nodes

**... What platforms does circuits support?** circuits currently supports Linux, FreeBSD, OSX and Windows and is currently continually tested against Linux and Windows against Python versions 2.6, 2.7, 3.1 and 3.2

**... Can circuits be used for concurrent or distributed programming?** Yes. We also have plans to build more distributed components into circuits making distributing computing with circuits very trivial.

Got more questions?

- Send an email to our Mailing List.
- Talk to us online on the #circuits IRC Channel

## 2.11 Glossary

**VCS** Version Control System, what you use for versioning your source code

## 2.12 Hello

Download Source Code: `hello.py`:

## 2.13 Echo Server

Download Source Code: `echoserver.py`:

## 2.14 Hello Web

Download Source Code: `helloweb.py`:

More examples...

## 2.15 Documentation TODO

## 2.16 PyPi README Page

circuits is a **Lightweight Event** driven and **Asynchronous Application Framework** for the Python Programming Language with a strong **Component** Architecture.

circuits also includes a lightweight, high performance and scalable HTTP/WSGI compliant web server as well as various I/O and Networking components.

- Website
- Downloads
- Documentation

Got questions?

- Ask a Question (Tag it: `circuits-framework`)

### 2.16.1 Examples

#### Hello

```
#!/usr/bin/env python

"""circuits Hello World"""

from circuits import Component, Event


class hello(Event):

    """hello Event"""


class App(Component):

    def hello(self):
        """Hello Event Handler"""

        print("Hello World!")

```

```
    def started(self, component):
        """Started Event Handler

        This is fired internally when your application starts up and can be used to
        trigger events that only occur once during startup.
        """

        self.fire(hello())  # Fire hello Event

        raise SystemExit(0)  # Terminate the Application

App().run()
```

Download Source Code: `hello.py`:

### Echo Server

```
#!/usr/bin/env python

"""Simple TCP Echo Server

This example shows how you can create a simple TCP Server (an Echo Service)
utilizing the builtin Socket Components that the circuits library ships with.
"""

from circuits import handler, Debugger
from circuits.net.sockets import TCPServer


class EchoServer(TCPServer):

    @handler("read")
    def on_read(self, sock, data):
        """Read Event Handler

        This is fired by the underlying Socket Component when there has been
        new data read from the connected client.

        ..note :: By simply returning, client/server socket components listen
                  to ValueChagned events (feedback) to determine if a handler
                  returned some data and fires a subsequent Write event with
                  the value returned.
        """

        return data

# Start and "run" the system.
# Bind to port 0.0.0.0:8000
app = EchoServer(("0.0.0.0", 8000))
Debugger().register(app)
app.run()
```

Download Source Code: `echoserver.py`:

#### Hello Web

```
#!/usr/bin/env python

from circuits.web import Server, Controller


class Root(Controller):

    def index(self):
        """Index Request Handler

        Controller(s) expose implicitly methods as request handlers.
        Request Handlers can still be customized by using the ``@expose``
        decorator. For example exposing as a different path.
        """

        return "Hello World!"

app = Server(("0.0.0.0", 8000))
Root().register(app)
app.run()
```

Download Source Code: `helloweb.py`:

More examples...

### 2.16.2 Features

- event driven
- concurrency support
- component architecture
- asynchronous I/O components
- no required external dependencies
- full featured web framework (circuits.web)
- coroutine based synchronization primitives

### 2.16.3 Requirements

- circuits has no dependencies beyond the Python Standard Library.

### 2.16.4 Supported Platforms

- Linux, FreeBSD, Mac OS X, Windows
- Python 2.6, 2.7, 3.2, 3.3, 3.4
- pypy 2.0, 2.1, 2.2

### 2.16.5 Installation

The simplest and recommended way to install circuits is with pip. You may install the latest stable release from PyPI with pip:

```
$ pip install circuits
```

If you do not have pip, you may use easy_install:

```
$ easy_install circuits
```

Alternatively, you may download the source package from the PyPi or the Downloads extract it and install using:

```
$ python setup.py install
```

**Note:** You can install the development version via `pip install circuits==dev`.

### 2.16.6 License

circuits is licensed under the MIT License.

### 2.16.7 Feedback

We welcome any questions or feedback about bugs and suggestions on how to improve circuits.

Let us know what you think about circuits. @pythoncircuits.

Do you have suggestions for improvement? Then please Create an Issue with details of what you would like to see. I'll take a look at it and work with you to either incorporate the idea or find a better solution.

### 2.16.8 Community

There are also several places you can reach out to the circuits community:

- Mailing List
- Google+ Group
- #circuits IRC Channel on the FreeNode IRC Network
- Ask a Question on Stackoverflow (Tag it: `circuits-framework`)

CHAPTER 3

# Indices and tables

- Index
- modindex
- search
- Glossary
- Documentation TODO
- PyPi README Page

## C

circuits, 43
circuits.app, 133
circuits.app.daemon, 134
circuits.core, 150
circuits.core.bridge, 157
circuits.core.components, 21
circuits.core.debugger, 23
circuits.core.events, 160
circuits.core.handlers, 163
circuits.core.helpers, 164
circuits.core.loader, 164
circuits.core.manager, 29
circuits.core.pollers, 167
circuits.core.timers, 169
circuits.core.utils, 169
circuits.core.values, 29
circuits.core.workers, 170
circuits.io, 185
circuits.io.events, 185
circuits.io.file, 198
circuits.io.process, 198
circuits.io.serial, 199
circuits.net, 205
circuits.net.events, 205
circuits.net.sockets, 209
circuits.node, 215
circuits.node.client, 216
circuits.node.events, 216
circuits.node.node, 217
circuits.node.server, 218
circuits.node.utils, 220
circuits.protocols, 223
circuits.protocols.http, 224
circuits.protocols.irc, 225
circuits.protocols.line, 225
circuits.protocols.websocket, 227
circuits.six, 227
circuits.tools, 228
circuits.version, 229
circuits.web, 35
circuits.web.client, 256
circuits.web.constants, 257
circuits.web.controllers, 257
circuits.web.dispatchers, 261
circuits.web.dispatchers.dispatcher, 261
circuits.web.dispatchers.jsonrpc, 261
circuits.web.dispatchers.static, 262
circuits.web.dispatchers.virtualhosts, 263
circuits.web.dispatchers.xmlrpc, 263
circuits.web.errors, 264
circuits.web.events, 265
circuits.web.exceptions, 268
circuits.web.headers, 272
circuits.web.http, 273
circuits.web.loggers, 274
circuits.web.main, 274
circuits.web.parsers, 278
circuits.web.parsers.http, 278
circuits.web.parsers.multipart, 279
circuits.web.parsers.querystring, 281
circuits.web.processors, 281
circuits.web.servers, 281
circuits.web.sessions, 283
circuits.web.tools, 283
circuits.web.url, 285
circuits.web.utils, 286
circuits.web.websockets, 288
circuits.web.websockets.client, 288
circuits.web.websockets.dispatcher, 288
circuits.web.wrappers, 289
circuits.web.wsgi, 290

## A

absolute() (circuits.web.url.URL method), 125, 253, 285
abspath() (circuits.web.url.URL method), 125, 253, 285
AcceptElement (class in circuits.web.headers), 119, 246, 272
accessed (class in circuits.io.events), 74, 171, 185
add() (circuits.node.node.Node method), 96, 213, 218
add_header() (circuits.web.headers.Headers method), 120, 247, 273
add_move() (in module circuits.six), 129, 227
addHandler() (circuits.core.Manager method), 72, 155
addHandler() (circuits.core.manager.Manager method), 63, 145, 166
addReader() (circuits.core.pollers.BasePoller method), 64, 147, 168
addReader() (circuits.core.pollers.EPoll method), 65, 148, 168
addReader() (circuits.core.pollers.KQueue method), 65, 148, 168
addReader() (circuits.core.pollers.Poll method), 65, 148, 168
addWriter() (circuits.core.pollers.BasePoller method), 64, 147, 168
addWriter() (circuits.core.pollers.EPoll method), 65, 148, 168
addWriter() (circuits.core.pollers.KQueue method), 65, 148, 168
addWriter() (circuits.core.pollers.Poll method), 65, 148, 168
alert_done (circuits.core.Event attribute), 70, 153
alert_done (circuits.core.events.Event attribute), 57, 140, 160
append() (circuits.web.headers.Headers method), 120, 247, 273
append() (circuits.web.parsers.multipart.MultiDict method), 107, 233, 276, 280
Application (class in circuits.web.wsgi), 128, 256, 290
ARRAY (circuits.web.parsers.querystring.QueryStringToken attribute), 108, 235, 278, 281
Authentication (class in circuits.web.main), 121, 248, 274
average() (in module circuits.web.utils), 126, 254, 286

## B

b() (in module circuits.six), 130, 228
BadGateway, 116, 243, 269
BadRequest, 117, 244, 270
base (circuits.web.http.HTTP attribute), 121, 248, 273
BaseComponent (class in circuits.core), 68, 151
BaseComponent (class in circuits.core.components), 55, 137, 158
BaseController (class in circuits.web.controllers), 110, 237, 257
BasePoller (class in circuits.core.pollers), 64, 147, 167
BaseServer (class in circuits.web.servers), 122, 249, 281
basic_auth() (in module circuits.web.tools), 124, 251, 283
body (circuits.web.wrappers.Response attribute), 128, 255, 290
Body (class in circuits.web.wrappers), 127, 254, 289
Bridge (class in circuits.core), 71, 154
Bridge (class in circuits.core.bridge), 54, 137, 157
broadcast (class in circuits.net.events), 88, 199, 205
broadcast() (circuits.net.sockets.UDPServer method), 93, 204, 211
byteindex() (in module circuits.six), 129, 227
bytes_to_str() (in module circuits.six), 130, 228

## C

call() (circuits.core.Manager method), 72, 155
call() (circuits.core.manager.Manager method), 63, 146, 167
callEvent() (circuits.core.Manager method), 72, 156
callEvent() (circuits.core.manager.Manager method), 63, 146, 167
CallValue (class in circuits.core.manager), 62, 144, 165
cancel() (circuits.core.Event method), 70, 153
cancel() (circuits.core.events.Event method), 57, 140, 161
canonical() (circuits.web.url.URL method), 126, 253, 285
CaseInsensitiveDict (class in circuits.web.headers), 119, 246, 272
channel (circuits.app.Daemon attribute), 54, 133

channel (circuits.app.daemon.Daemon attribute), 51, 131, 134
channel (circuits.core.BaseComponent attribute), 68, 152
channel (circuits.core.Bridge attribute), 71, 154
channel (circuits.core.bridge.Bridge attribute), 54, 137, 157
channel (circuits.core.components.BaseComponent attribute), 55, 138, 158
channel (circuits.core.loader.Loader attribute), 61, 144, 165
channel (circuits.core.pollers.BasePoller attribute), 64, 147, 167
channel (circuits.core.pollers.EPoll attribute), 65, 148, 168
channel (circuits.core.pollers.KQueue attribute), 65, 148, 168
channel (circuits.core.pollers.Poll attribute), 65, 148, 168
channel (circuits.core.pollers.Select attribute), 65, 147, 168
channel (circuits.core.Worker attribute), 71, 154
channel (circuits.core.workers.Worker attribute), 67, 150, 171
channel (circuits.io.file.File attribute), 87, 184, 198
channel (circuits.io.process.Process attribute), 87, 184, 198
channel (circuits.io.serial.Serial attribute), 88, 185, 199
channel (circuits.net.sockets.Client attribute), 92, 203, 209
channel (circuits.net.sockets.Server attribute), 92, 204, 210
channel (circuits.node.client.Client attribute), 94, 212, 216
channel (circuits.node.node.Node attribute), 96, 214, 218
channel (circuits.node.server.Server attribute), 97, 214, 219
channel (circuits.protocols.http.HTTP attribute), 98, 220, 224
channel (circuits.protocols.websocket.WebSocketCodec attribute), 102, 223, 227
channel (circuits.web.client.Client attribute), 110, 236, 256
channel (circuits.web.controllers.BaseController attribute), 110, 237, 257
channel (circuits.web.dispatchers.dispatcher.Dispatcher attribute), 103, 229, 258, 261
channel (circuits.web.dispatchers.jsonrpc.JSONRPC attribute), 103, 229, 259, 262
channel (circuits.web.dispatchers.static.Static attribute), 104, 230, 260, 262
channel (circuits.web.dispatchers.virtualhosts.VirtualHosts attribute), 104, 231, 260, 263
channel (circuits.web.dispatchers.xmlrpc.XMLRPC attribute), 104, 231, 260, 263
channel (circuits.web.http.HTTP attribute), 121, 248, 273
channel (circuits.web.loggers.Logger attribute), 121, 248, 274
channel (circuits.web.main.Authentication attribute), 121, 248, 274
channel (circuits.web.main.HelloWorld attribute), 121, 248, 274
channel (circuits.web.servers.BaseServer attribute), 122, 249, 282
channel (circuits.web.servers.StdinServer attribute), 123, 250, 282
channel (circuits.web.sessions.Sessions attribute), 123, 250, 283
channel (circuits.web.websockets.client.WebSocketClient attribute), 109, 235, 287, 288
channel (circuits.web.websockets.dispatcher.WebSocketsDispatcher attribute), 109, 236, 288, 289
channel (circuits.web.wsgi.Application attribute), 128, 256, 290
channel (circuits.web.wsgi.Gateway attribute), 129, 256, 291
channels (circuits.core.Event attribute), 70, 153
channels (circuits.core.events.Event attribute), 57, 140, 160
check_auth() (in module circuits.web.tools), 124, 251, 284
child() (circuits.core.Event method), 70, 153
child() (circuits.core.events.Event method), 57, 140, 161
chunked (circuits.web.wrappers.Response attribute), 128, 255, 290
circuits (module), 43, 130
circuits.app (module), 53, 133
circuits.app.daemon (module), 51, 130, 134
circuits.core (module), 67, 150
circuits.core.bridge (module), 54, 137, 157
circuits.core.components (module), 21, 55, 137, 158
circuits.core.debugger (module), 23, 56, 139, 159
circuits.core.events (module), 56, 139, 160
circuits.core.handlers (module), 60, 143, 163
circuits.core.helpers (module), 61, 144, 164
circuits.core.loader (module), 61, 144, 164
circuits.core.manager (module), 29, 61, 144, 165
circuits.core.pollers (module), 64, 147, 167
circuits.core.timers (module), 65, 148, 169
circuits.core.utils (module), 66, 149, 169
circuits.core.values (module), 29, 66, 149, 170
circuits.core.workers (module), 67, 150, 170
circuits.io (module), 88, 185
circuits.io.events (module), 74, 171, 185
circuits.io.file (module), 87, 184, 198
circuits.io.process (module), 87, 184, 198
circuits.io.serial (module), 88, 185, 199
circuits.net (module), 94, 205
circuits.net.events (module), 88, 199, 205
circuits.net.sockets (module), 92, 203, 209

circuits.node (module), 98, 215
circuits.node.client (module), 94, 211, 216
circuits.node.events (module), 95, 212, 216
circuits.node.node (module), 95, 213, 217
circuits.node.server (module), 97, 214, 218
circuits.node.utils (module), 98, 215, 220
circuits.protocols (module), 102, 223
circuits.protocols.http (module), 98, 220, 224
circuits.protocols.irc (module), 100, 221, 225
circuits.protocols.line (module), 100, 222, 225
circuits.protocols.websocket (module), 101, 223, 227
circuits.six (module), 129, 227
circuits.tools (module), 102, 228
circuits.version (module), 130, 229
circuits.web (module), 35, 129, 256
circuits.web.client (module), 110, 236, 256
circuits.web.constants (module), 110, 237, 257
circuits.web.controllers (module), 110, 237, 257
circuits.web.dispatchers (module), 105, 232, 261
circuits.web.dispatchers.dispatcher (module), 103, 229, 258, 261
circuits.web.dispatchers.jsonrpc (module), 103, 229, 259, 261
circuits.web.dispatchers.static (module), 104, 230, 259, 262
circuits.web.dispatchers.virtualhosts (module), 104, 230, 260, 263
circuits.web.dispatchers.xmlrpc (module), 104, 231, 260, 263
circuits.web.errors (module), 111, 238, 264
circuits.web.events (module), 112, 239, 265
circuits.web.exceptions (module), 115, 242, 268
circuits.web.headers (module), 119, 246, 272
circuits.web.http (module), 121, 247, 273
circuits.web.loggers (module), 121, 248, 274
circuits.web.main (module), 121, 248, 274
circuits.web.parsers (module), 108, 235, 278
circuits.web.parsers.http (module), 105, 232, 275, 278
circuits.web.parsers.multipart (module), 106, 233, 276, 279
circuits.web.parsers.querystring (module), 108, 235, 278, 281
circuits.web.processors (module), 122, 249, 281
circuits.web.servers (module), 122, 249, 281
circuits.web.sessions (module), 123, 250, 283
circuits.web.tools (module), 124, 251, 283
circuits.web.url (module), 125, 252, 285
circuits.web.utils (module), 126, 254, 286
circuits.web.websockets (module), 110, 236, 288
circuits.web.websockets.client (module), 109, 235, 287, 288
circuits.web.websockets.dispatcher (module), 109, 236, 287, 288
circuits.web.wrappers (module), 127, 254, 289
circuits.web.wsgi (module), 128, 256, 290
Client (class in circuits.net.sockets), 92, 203, 209
Client (class in circuits.node.client), 94, 211, 216
Client (class in circuits.web.client), 110, 236, 256
close (circuits.web.wrappers.Response attribute), 128, 255, 290
close (class in circuits.io.events), 75, 172, 186
close (class in circuits.net.events), 88, 199, 206
close() (circuits.io.file.File method), 87, 184, 198
close() (circuits.io.serial.Serial method), 88, 185, 199
close() (circuits.net.sockets.Client method), 92, 203, 209
close() (circuits.net.sockets.Server method), 92, 204, 210
close() (circuits.net.sockets.UDPServer method), 93, 205, 211
close() (circuits.node.client.Client method), 94, 212, 216
close() (circuits.web.client.Client method), 110, 236, 256
close() (circuits.web.websockets.client.WebSocketClient method), 109, 236, 287, 288
closed (circuits.io.file.File attribute), 87, 184, 198
closed (class in circuits.io.events), 75, 172, 187
closed (class in circuits.net.events), 89, 200, 206
code (circuits.web.errors.forbidden attribute), 111, 238, 264
code (circuits.web.errors.httperror attribute), 112, 238, 264
code (circuits.web.errors.notfound attribute), 112, 238, 264
code (circuits.web.errors.unauthorized attribute), 112, 239, 265
code (circuits.web.exceptions.BadGateway attribute), 117, 243, 269
code (circuits.web.exceptions.BadRequest attribute), 117, 244, 270
code (circuits.web.exceptions.Forbidden attribute), 119, 245, 271
code (circuits.web.exceptions.Gone attribute), 118, 244, 270
code (circuits.web.exceptions.HTTPException attribute), 117, 243, 269
code (circuits.web.exceptions.InternalServerError attribute), 118, 245, 271
code (circuits.web.exceptions.LengthRequired attribute), 119, 245, 271
code (circuits.web.exceptions.MethodNotAllowed attribute), 118, 245, 271
code (circuits.web.exceptions.NotAcceptable attribute), 117, 244, 270
code (circuits.web.exceptions.NotFound attribute), 118, 245, 271
code (circuits.web.exceptions.NotImplemented attribute), 119, 246, 272
code (circuits.web.exceptions.PreconditionFailed attribute), 118, 245, 270

code (circuits.web.exceptions.RangeUnsatisfiable attribute), 117, 244, 270
code (circuits.web.exceptions.Redirect attribute), 116, 242, 268
code (circuits.web.exceptions.RequestEntityTooLarge attribute), 116, 243, 268
code (circuits.web.exceptions.RequestTimeout attribute), 116, 242, 268
code (circuits.web.exceptions.RequestURITooLarge attribute), 116, 243, 269
code (circuits.web.exceptions.ServiceUnavailable attribute), 116, 243, 269
code (circuits.web.exceptions.Unauthorized attribute), 117, 244, 270
code (circuits.web.exceptions.UnsupportedMediaType attribute), 116, 243, 269
complete (circuits.core.components.prepare_unregister attribute), 55, 137, 158
complete (circuits.core.Event attribute), 70, 153
complete (circuits.core.events.Event attribute), 57, 140, 160
complete (circuits.web.events.request attribute), 113, 240, 266
complete (circuits.web.events.response attribute), 114, 240, 266
complete (circuits.web.events.stream attribute), 115, 241, 267
Component (class in circuits.core), 69, 152
Component (class in circuits.core.components), 56, 138, 159
compress() (in module circuits.web.utils), 126, 254, 286
connect (class in circuits.net.events), 89, 201, 207
connect() (circuits.net.sockets.TCPClient method), 93, 204, 210
connect() (circuits.net.sockets.UNIXClient method), 93, 205, 211
connect() (circuits.node.client.Client method), 94, 212, 216
connect() (circuits.web.client.Client method), 110, 236, 256
connected (circuits.net.sockets.Client attribute), 92, 203, 209
connected (circuits.net.sockets.Server attribute), 92, 204, 210
connected (circuits.web.client.Client attribute), 110, 236, 256
connected (circuits.web.websockets.client.WebSocketClient attribute), 109, 236, 287, 288
connected (class in circuits.net.events), 90, 201, 207
connected_to (class in circuits.node.events), 95, 212, 216
Controller (class in circuits.web.controllers), 111, 238, 258
copy_file() (in module circuits.web.parsers.multipart), 108, 234, 277, 280
create() (circuits.core.Event class method), 70, 153
create() (circuits.core.events.Event class method), 57, 140, 160
create_bound_method() (in module circuits.six), 129, 227
create_environ() (in module circuits.web.wsgi), 129, 256, 291
create_session() (in module circuits.web.sessions), 123, 250, 283
created (class in circuits.io.events), 76, 173, 187

## D

Daemon (class in circuits.app), 53, 133
Daemon (class in circuits.app.daemon), 51, 130, 134
daemonize (class in circuits.app.daemon), 51, 131, 134
daemonize() (circuits.app.Daemon method), 54, 133
daemonize() (circuits.app.daemon.Daemon method), 51, 131, 134
Debugger (class in circuits.core), 71, 154
Debugger (class in circuits.core.debugger), 56, 139, 159
defrag() (circuits.web.url.URL method), 126, 253, 285
deleted (class in circuits.io.events), 77, 174, 188
deletepid (class in circuits.app.daemon), 52, 131, 135
deletepid() (circuits.app.Daemon method), 54, 133
deletepid() (circuits.app.daemon.Daemon method), 51, 131, 134
deparam() (circuits.web.url.URL method), 126, 253, 285
deprecated() (in module circuits.tools), 102, 229
description (circuits.web.errors.httperror attribute), 112, 238, 264
description (circuits.web.exceptions.BadGateway attribute), 117, 243, 269
description (circuits.web.exceptions.BadRequest attribute), 117, 244, 270
description (circuits.web.exceptions.Forbidden attribute), 119, 246, 271
description (circuits.web.exceptions.Gone attribute), 118, 244, 270
description (circuits.web.exceptions.HTTPException attribute), 117, 243, 269
description (circuits.web.exceptions.InternalServerError attribute), 118, 245, 271
description (circuits.web.exceptions.LengthRequired attribute), 119, 245, 271
description (circuits.web.exceptions.NotAcceptable attribute), 117, 244, 270
description (circuits.web.exceptions.NotFound attribute), 118, 245, 271
description (circuits.web.exceptions.NotImplemented attribute), 119, 246, 272
description (circuits.web.exceptions.PreconditionFailed attribute), 118, 245, 271
description (circuits.web.exceptions.RangeUnsatisfiable attribute), 117, 244, 270

description (circuits.web.exceptions.RequestEntityTooLarge attribute), 116, 243, 269
description (circuits.web.exceptions.RequestTimeout attribute), 116, 242, 268
description (circuits.web.exceptions.RequestURITooLarge attribute), 116, 243, 269
description (circuits.web.exceptions.ServiceUnavailable attribute), 116, 243, 269
description (circuits.web.exceptions.Unauthorized attribute), 117, 244, 270
description (circuits.web.exceptions.UnsupportedMediaType attribute), 116, 243, 269
dictform() (in module circuits.web.utils), 126, 254, 286
digest_auth() (in module circuits.web.tools), 124, 251, 284
discard() (circuits.core.pollers.BasePoller method), 64, 147, 168
discard() (circuits.core.pollers.EPoll method), 65, 148, 168
discard() (circuits.core.pollers.KQueue method), 65, 148, 169
discard() (circuits.core.pollers.Poll method), 65, 148, 168
disconnect (class in circuits.net.events), 90, 201, 207
disconnected (class in circuits.net.events), 90, 201, 208
disconnected_from (class in circuits.node.events), 95, 212, 217
Dispatcher (class in circuits.web.dispatchers.dispatcher), 103, 229, 258, 261
display_banner (circuits.web.servers.BaseServer attribute), 122, 249, 282
do_handshake() (in module circuits.net.sockets), 93, 205, 211
done (circuits.web.wrappers.Response attribute), 128, 255, 290
drop_privileges() (circuits.app.DropPrivileges method), 54, 134
DropPrivileges (class in circuits.app), 54, 133
dump_event() (in module circuits.node.utils), 98, 215, 220
dump_value() (in module circuits.node.utils), 98, 215, 220

## E

edges() (in module circuits.tools), 102, 228
elements() (circuits.web.headers.Headers method), 120, 247, 273
encode() (circuits.web.url.URL method), 126, 253, 285
eof (class in circuits.io.events), 78, 175, 189
EPoll (class in circuits.core.pollers), 65, 148, 168
equiv() (circuits.web.url.URL method), 126, 253, 285
error (class in circuits.io.events), 78, 175, 189
error (class in circuits.net.events), 91, 202, 208
escape() (circuits.web.url.URL method), 126, 253, 285
Event (class in circuits.core), 69, 152
Event (class in circuits.core.events), 56, 139, 160
events() (circuits.core.BaseComponent class method), 68, 152
events() (circuits.core.components.BaseComponent class method), 56, 138, 159
EventType (class in circuits.core.events), 56, 139, 160
exception (class in circuits.core.events), 57, 140, 161
ExceptionWrapper (class in circuits.core.manager), 62, 144, 165
exec_() (in module circuits.six), 130, 228
execute() (circuits.web.parsers.http.HttpParser method), 105, 232, 275, 278
expired (circuits.core.manager.Sleep attribute), 62, 145, 165
expires() (circuits.web.controllers.BaseController method), 110, 237, 257
expires() (in module circuits.web.tools), 124, 251, 284
expiry (circuits.core.Timer attribute), 72, 155
expiry (circuits.core.timers.Timer attribute), 66, 149, 169
expose() (in module circuits.web.controllers), 111, 238, 258
exposeJSON() (in module circuits.web.controllers), 111, 238, 258
ExposeJSONMetaClass (class in circuits.web.controllers), 111, 238, 258
ExposeMetaClass (class in circuits.web.controllers), 111, 238, 258
extract() (circuits.core.manager.ExceptionWrapper method), 62, 145, 165

## F

failure (circuits.core.Event attribute), 70, 153
failure (circuits.core.events.Event attribute), 57, 140, 160
failure (circuits.core.task attribute), 70, 153
failure (circuits.core.workers.task attribute), 67, 150, 170
failure (circuits.web.events.request attribute), 113, 240, 266
failure (circuits.web.events.response attribute), 114, 240, 266
failure (circuits.web.events.stream attribute), 115, 241, 267
FakeSock (class in circuits.web.servers), 123, 250, 282
FallBackExceptionHandler (class in circuits.core.helpers), 61, 144, 164
FallBackGenerator (class in circuits.core.helpers), 61, 144, 164
FallBackSignalHandler (class in circuits.core.helpers), 61, 144, 164
feed() (circuits.web.parsers.multipart.MultipartPart method), 107, 234, 277, 280
File (class in circuits.io.file), 87, 184, 198
file_generator() (in module circuits.web.wrappers), 128, 256, 290
filename (circuits.io.file.File attribute), 87, 184, 198

find_handlers() (in module circuits.web.dispatchers.dispatcher), 103, 229, 258, 261
findchannel() (in module circuits.core.utils), 66, 149, 169
findcmp() (in module circuits.core.utils), 66, 149, 169
findroot() (in module circuits.core.utils), 66, 149, 169
findroot() (in module circuits.tools), 102, 228
findtype() (in module circuits.core.utils), 66, 149, 169
finish_header() (circuits.web.parsers.multipart.MultipartPart method), 107, 234, 277, 280
fire() (circuits.core.Manager method), 72, 156
fire() (circuits.core.manager.Manager method), 63, 146, 166
fireEvent() (circuits.core.Manager method), 73, 156
fireEvent() (circuits.core.manager.Manager method), 63, 146, 166
flatten() (in module circuits.core.utils), 66, 149, 169
flush() (circuits.core.Manager method), 73, 156
flush() (circuits.core.manager.Manager method), 63, 146, 167
flushEvents() (circuits.core.Manager method), 73, 156
flushEvents() (circuits.core.manager.Manager method), 63, 146, 167
Forbidden, 119, 245, 271
forbidden (class in circuits.web.errors), 111, 238, 264
forbidden() (circuits.web.controllers.BaseController method), 110, 237, 257
format (circuits.web.loggers.Logger attribute), 121, 248, 274
formattime() (in module circuits.web.loggers), 121, 248, 274
from_str() (circuits.web.headers.AcceptElement class method), 119, 246, 272
from_str() (circuits.web.headers.HeaderElement class method), 120, 246, 272
fromkeys() (circuits.web.headers.CaseInsensitiveDict class method), 119, 246, 272

## G

Gateway (class in circuits.web.wsgi), 129, 256, 291
generate_events (class in circuits.core.events), 59, 142, 162
get() (circuits.web.headers.CaseInsensitiveDict method), 119, 246, 272
get() (circuits.web.parsers.multipart.MultiDict method), 107, 233, 276, 280
get() (circuits.web.parsers.multipart.MultipartParser method), 107, 234, 277, 280
get_all() (circuits.web.headers.Headers method), 120, 247, 273
get_all() (circuits.web.parsers.multipart.MultipartParser method), 107, 234, 277, 280
get_connection_names() (circuits.node.node.Node method), 96, 214, 218
get_headers() (circuits.web.parsers.http.HttpParser method), 105, 232, 275, 278
get_method() (circuits.web.parsers.http.HttpParser method), 105, 232, 275, 278
get_path() (circuits.web.parsers.http.HttpParser method), 105, 232, 275, 278
get_peer() (circuits.node.node.Node method), 97, 214, 218
get_query_string() (circuits.web.parsers.http.HttpParser method), 105, 232, 275, 278
get_ranges() (in module circuits.web.utils), 126, 254, 286
get_scheme() (circuits.web.parsers.http.HttpParser method), 105, 232, 275, 278
get_socks() (circuits.node.server.Server method), 97, 214, 219
get_status_code() (circuits.web.parsers.http.HttpParser method), 105, 232, 275, 278
get_unbound_function() (in module circuits.six), 129, 227
get_url() (circuits.web.parsers.http.HttpParser method), 105, 232, 275, 278
get_version() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 278
getall() (circuits.web.parsers.multipart.MultiDict method), 107, 233, 276, 280
getHandlers() (circuits.core.Manager method), 73, 156
getHandlers() (circuits.core.manager.Manager method), 63, 145, 166
getpeername() (circuits.web.servers.FakeSock method), 123, 250, 282
getRequestResponse() (circuits.web.wsgi.Application method), 128, 256, 290
getTarget() (circuits.core.pollers.BasePoller method), 64, 147, 168
getValue() (circuits.core.values.Value method), 66, 149, 170
Gone, 118, 244, 270
graph() (in module circuits.tools), 102, 228
gzip() (in module circuits.web.tools), 125, 252, 284

## H

handled (circuits.web.wrappers.Request attribute), 128, 255, 290
handler() (in module circuits.core), 67, 150
handler() (in module circuits.core.handlers), 60, 143, 163
HandlerMetaClass (class in circuits.core.handlers), 61, 143, 164
handlers() (circuits.core.BaseComponent class method), 68, 152
handlers() (circuits.core.components.BaseComponent class method), 56, 138, 159
handles() (circuits.core.BaseComponent class method), 69, 152

handles() (circuits.core.components.BaseComponent class method), 56, 138, 159
header_elements() (in module circuits.web.headers), 120, 247, 273
header_quote() (in module circuits.web.parsers.multipart), 108, 234, 277, 280
header_unquote() (in module circuits.web.parsers.multipart), 108, 234, 277, 281
HeaderElement (class in circuits.web.headers), 119, 246, 272
headerNames (circuits.web.wsgi.Application attribute), 128, 256, 291
headers (circuits.web.tools.ReverseProxy attribute), 124, 251, 283
Headers (class in circuits.web.headers), 120, 246, 272
hello() (circuits.web.main.Root method), 122, 249, 274
HelloWorld (class in circuits.web.main), 121, 248, 274
host (circuits.net.sockets.Server attribute), 92, 204, 210
host (circuits.node.server.Server attribute), 97, 215, 219
host (circuits.web.servers.BaseServer attribute), 122, 249, 282
host (circuits.web.servers.StdinServer attribute), 123, 250, 282
host (circuits.web.wrappers.Request attribute), 128, 255, 290
host (circuits.web.wsgi.Application attribute), 129, 256, 291
Host (class in circuits.web.wrappers), 127, 254, 289
HTTP (class in circuits.protocols.http), 98, 220, 224
HTTP (class in circuits.web.http), 121, 247, 273
httperror (class in circuits.web.errors), 111, 238, 264
HTTPException, 110, 117, 236, 243, 257, 269
HttpParser (class in circuits.web.parsers.http), 105, 232, 275, 278
HTTPStatus (class in circuits.web.wrappers), 127, 254, 289

## I

IgnoreChannels (circuits.core.Debugger attribute), 71, 154
IgnoreChannels (circuits.core.debugger.Debugger attribute), 56, 139, 159
IgnoreEvents (circuits.core.Debugger attribute), 71, 154
IgnoreEvents (circuits.core.debugger.Debugger attribute), 56, 139, 159
in_subtree() (circuits.core.components.prepare_unregister method), 55, 137, 158
index (circuits.web.wrappers.Request attribute), 128, 255, 290
inform() (circuits.core.values.Value method), 66, 149, 170
init() (circuits.app.Daemon method), 54, 133
init() (circuits.app.daemon.Daemon method), 51, 131, 134
init() (circuits.app.DropPrivileges method), 54, 134
init() (circuits.core.Bridge method), 71, 154
init() (circuits.core.bridge.Bridge method), 54, 137, 158
init() (circuits.core.Worker method), 71, 154
init() (circuits.core.workers.Worker method), 67, 150, 171
init() (circuits.io.file.File method), 87, 184, 198
init() (circuits.io.process.Process method), 87, 184, 198
init() (circuits.net.sockets.TCPClient method), 93, 204, 210
init() (circuits.web.tools.ReverseProxy method), 124, 251, 283
init() (circuits.web.wsgi.Application method), 129, 256, 291
init() (circuits.web.wsgi.Gateway method), 129, 256, 291
inspect() (in module circuits.tools), 102, 229
InternalServerError, 118, 245, 271
InvalidChunkSize, 106, 232, 275, 279
InvalidHeader, 106, 232, 275, 279
InvalidRequestLine, 106, 233, 275, 279
ip (circuits.web.wrappers.Host attribute), 127, 254, 289
ipc (class in circuits.core), 71, 154
ipc (class in circuits.core.bridge), 54, 137, 157
is_buffered() (circuits.web.parsers.multipart.MultipartPart method), 108, 234, 277, 280
is_chunked() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 278
is_headers_complete() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 278
is_message_begin() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 278
is_message_complete() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 278
is_partial_body() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 278
is_upgrade() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 278
isReading() (circuits.core.pollers.BasePoller method), 64, 147, 168
isWriting() (circuits.core.pollers.BasePoller method), 64, 147, 168
items() (circuits.web.headers.Headers method), 120, 247, 273
iterallitems() (circuits.web.parsers.multipart.MultiDict method), 107, 233, 276, 280
Iterator (class in circuits.six), 129, 228
iterbytes() (in module circuits.six), 129, 227
iteritems() (in module circuits.six), 130, 228
iterkeys() (in module circuits.six), 129, 228
itervalues() (in module circuits.six), 129, 228

## J

join() (circuits.core.Manager method), 73, 156
join() (circuits.core.manager.Manager method), 64, 146, 167
JSONController (class in circuits.web.controllers), 111, 238, 258
JSONRPC (class in circuits.web.dispatchers.jsonrpc), 103, 229, 259, 261

## K

KEY (circuits.web.parsers.querystring.QueryStringToken attribute), 108, 235, 278, 281
keys() (circuits.web.parsers.multipart.MultiDict method), 107, 233, 276, 280
kill() (circuits.io.process.Process method), 87, 184, 199
kill() (in module circuits.tools), 102, 228
KQueue (class in circuits.core.pollers), 65, 148, 168

## L

LengthRequired, 118, 245, 271
Line (class in circuits.protocols.line), 100, 222, 225
line (class in circuits.protocols.line), 100, 222, 226
load() (circuits.core.loader.Loader method), 61, 144, 165
load() (circuits.web.sessions.Sessions method), 123, 250, 283
load_event() (in module circuits.node.utils), 98, 215, 220
load_value() (in module circuits.node.utils), 98, 215, 220
Loader (class in circuits.core.loader), 61, 144, 164
local (circuits.web.wrappers.Request attribute), 128, 255, 290
lock (circuits.core.events.generate_events attribute), 60, 142, 163
log() (circuits.web.loggers.Logger method), 121, 248, 274
log_response() (circuits.web.loggers.Logger method), 121, 248, 274
Logger (class in circuits.web.loggers), 121, 248, 274
login (circuits.web.wrappers.Request attribute), 128, 255, 290
lower() (circuits.web.url.URL method), 126, 253, 286

## M

main() (in module circuits.web.main), 122, 249, 274
Manager (class in circuits.core), 72, 155
Manager (class in circuits.core.manager), 62, 145, 165
MethodNotAllowed, 118, 245, 271
mode (circuits.io.file.File attribute), 87, 184, 198
modified (class in circuits.io.events), 79, 176, 190
moved (class in circuits.io.events), 80, 177, 191
MovedAttribute (class in circuits.six), 129, 227
MovedModule (class in circuits.six), 129, 227
MultiDict (class in circuits.web.parsers.multipart), 107, 233, 276, 279
MultipartError, 107, 233, 276, 280
MultipartParser (class in circuits.web.parsers.multipart), 107, 233, 276, 280
MultipartPart (class in circuits.web.parsers.multipart), 107, 234, 277, 280

## N

name (circuits.app.daemon.daemonize attribute), 52, 131, 135
name (circuits.app.daemon.deletepid attribute), 52, 132, 136
name (circuits.app.daemon.writepid attribute), 53, 133, 136
name (circuits.core.bridge.ipc attribute), 54, 137, 157
name (circuits.core.components.prepare_unregister attribute), 55, 137, 158
name (circuits.core.Event attribute), 70, 153
name (circuits.core.events.Event attribute), 57, 140, 161
name (circuits.core.events.exception attribute), 58, 140, 161
name (circuits.core.events.generate_events attribute), 60, 142, 163
name (circuits.core.events.registered attribute), 58, 141, 162
name (circuits.core.events.signal attribute), 58, 141, 161
name (circuits.core.events.started attribute), 58, 141, 161
name (circuits.core.events.stopped attribute), 58, 141, 161
name (circuits.core.events.unregistered attribute), 59, 142, 162
name (circuits.core.ipc attribute), 71, 154
name (circuits.core.Manager attribute), 73, 156
name (circuits.core.manager.Manager attribute), 62, 145, 166
name (circuits.core.task attribute), 70, 153
name (circuits.core.workers.task attribute), 67, 150, 171
name (circuits.io.events.accessed attribute), 75, 172, 186
name (circuits.io.events.close attribute), 75, 172, 186
name (circuits.io.events.closed attribute), 76, 173, 187
name (circuits.io.events.created attribute), 77, 174, 188
name (circuits.io.events.deleted attribute), 77, 174, 189
name (circuits.io.events.eof attribute), 78, 175, 189
name (circuits.io.events.error attribute), 79, 176, 190
name (circuits.io.events.modified attribute), 80, 177, 191
name (circuits.io.events.moved attribute), 80, 177, 191
name (circuits.io.events.open attribute), 81, 178, 192
name (circuits.io.events.opened attribute), 82, 179, 193
name (circuits.io.events.read attribute), 82, 179, 194
name (circuits.io.events.ready attribute), 83, 180, 194
name (circuits.io.events.seek attribute), 84, 181, 195
name (circuits.io.events.started attribute), 85, 182, 196
name (circuits.io.events.stopped attribute), 85, 182, 196
name (circuits.io.events.unmounted attribute), 86, 183, 197
name (circuits.io.events.write attribute), 87, 184, 198

name (circuits.net.events.broadcast attribute), 88, 199, 205
name (circuits.net.events.close attribute), 89, 200, 206
name (circuits.net.events.closed attribute), 89, 201, 207
name (circuits.net.events.connect attribute), 90, 201, 207
name (circuits.net.events.connected attribute), 90, 201, 207
name (circuits.net.events.disconnect attribute), 90, 201, 208
name (circuits.net.events.disconnected attribute), 90, 202, 208
name (circuits.net.events.error attribute), 91, 202, 208
name (circuits.net.events.read attribute), 91, 202, 208
name (circuits.net.events.ready attribute), 91, 203, 209
name (circuits.net.events.unreachable attribute), 92, 203, 209
name (circuits.net.events.write attribute), 92, 203, 209
name (circuits.node.events.connected_to attribute), 95, 212, 216
name (circuits.node.events.disconnected_from attribute), 95, 212, 217
name (circuits.node.events.remote attribute), 95, 213, 217
name (circuits.protocols.http.request attribute), 99, 221, 224
name (circuits.protocols.http.response attribute), 100, 221, 225
name (circuits.protocols.line.line attribute), 101, 223, 227
name (circuits.web.client.request attribute), 110, 237, 257
name (circuits.web.dispatchers.jsonrpc.rpc attribute), 104, 230, 259, 262
name (circuits.web.dispatchers.xmlrpc.rpc attribute), 105, 231, 261, 264
name (circuits.web.errors.forbidden attribute), 111, 238, 264
name (circuits.web.errors.httperror attribute), 112, 238, 264
name (circuits.web.errors.notfound attribute), 112, 239, 265
name (circuits.web.errors.redirect attribute), 112, 239, 265
name (circuits.web.errors.unauthorized attribute), 112, 239, 265
name (circuits.web.events.request attribute), 113, 240, 266
name (circuits.web.events.response attribute), 114, 241, 266
name (circuits.web.events.stream attribute), 115, 241, 267
name (circuits.web.events.terminate attribute), 115, 242, 268
name (circuits.web.exceptions.HTTPException attribute), 117, 243, 269
name (circuits.web.wrappers.Host attribute), 127, 254, 289
next() (circuits.six.Iterator method), 129, 228
Node (class in circuits.node.node), 95, 213, 217
NotAcceptable, 117, 244, 270
NotConnected, 110, 236, 257
NotFound, 118, 245, 271
notfound (class in circuits.web.errors), 112, 238, 264
notfound() (circuits.web.controllers.BaseController method), 111, 237, 257
notify (circuits.core.Event attribute), 70, 153
notify (circuits.core.events.Event attribute), 57, 140, 160
NotImplemented, 119, 246, 271

## O

OBJECT (circuits.web.parsers.querystring.QueryStringToken attribute), 108, 235, 278, 281
on_ready() (circuits.app.DropPrivileges method), 54, 134
on_started() (circuits.app.Daemon method), 54, 133
on_started() (circuits.app.daemon.Daemon method), 51, 131, 134
open (class in circuits.io.events), 80, 177, 191
opened (class in circuits.io.events), 81, 178, 192

## P

parent (circuits.core.Event attribute), 70, 153
parent (circuits.core.events.Event attribute), 57, 140, 160
parse() (circuits.web.headers.HeaderElement static method), 120, 246, 272
parse() (circuits.web.parsers.querystring.QueryStringParser method), 108, 235, 278, 281
parse() (circuits.web.url.URL class method), 126, 253, 286
parse_bind() (in module circuits.web.main), 122, 249, 274
parse_bind_parameter() (circuits.net.sockets.Client method), 92, 203, 209
parse_bind_parameter() (circuits.net.sockets.Server method), 92, 204, 210
parse_bind_parameter() (circuits.net.sockets.TCP6Client method), 93, 204, 210
parse_bind_parameter() (circuits.net.sockets.TCP6Server method), 93, 204, 210
parse_bind_parameter() (circuits.net.sockets.TCPServer method), 93, 204, 210
parse_bind_parameter() (circuits.net.sockets.UDP6Server method), 93, 204, 210
parse_body() (in module circuits.web.utils), 127, 254, 286
parse_form_data() (in module circuits.web.parsers.multipart), 108, 234, 277, 281
parse_ipv4_parameter() (in module circuits.net.sockets), 94, 205, 211
parse_ipv6_parameter() (in module circuits.net.sockets), 94, 205, 211

parse_options() (in module circuits.web.main), 122, 249, 274
parse_options_header() (in module circuits.web.parsers.multipart), 108, 235, 277, 281
parse_qs() (in module circuits.web.utils), 127, 254, 286
parse_url() (in module circuits.web.client), 110, 237, 257
parse_url() (in module circuits.web.url), 126, 253, 286
parts() (circuits.web.parsers.multipart.MultipartParser method), 107, 234, 277, 280
pid (circuits.core.Manager attribute), 73, 156
pid (circuits.core.manager.Manager attribute), 63, 145, 166
Pipe() (in module circuits.net.sockets), 92, 203, 209
Poll (class in circuits.core.pollers), 65, 147, 168
Poller (in module circuits.core.pollers), 65, 148, 169
pop() (circuits.web.headers.CaseInsensitiveDict method), 119, 246, 272
port (circuits.net.sockets.Server attribute), 92, 204, 210
port (circuits.node.server.Server attribute), 97, 215, 219
port (circuits.web.servers.BaseServer attribute), 122, 249, 282
port (circuits.web.servers.StdinServer attribute), 123, 250, 282
port (circuits.web.wrappers.Host attribute), 127, 254, 289
port (circuits.web.wsgi.Application attribute), 129, 256, 291
PreconditionFailed, 118, 244, 270
prepare() (circuits.web.wrappers.Response method), 128, 255, 290
prepare_unregister (class in circuits.core.components), 55, 137, 158
print_() (in module circuits.six), 130, 228
Process (class in circuits.io.process), 87, 184, 198
process() (circuits.web.parsers.querystring.QueryStringParser method), 108, 235, 278, 281
process() (in module circuits.web.processors), 122, 249, 281
process_multipart() (in module circuits.web.processors), 122, 249, 281
process_urlencoded() (in module circuits.web.processors), 122, 249, 281
processTask() (circuits.core.Manager method), 73, 156
processTask() (circuits.core.manager.Manager method), 64, 147, 167
protocol (circuits.web.http.HTTP attribute), 121, 248, 273
protocol (circuits.web.wrappers.Request attribute), 128, 255, 290
punycode() (circuits.web.url.URL method), 126, 253, 286

## Q

QueryStringParser (class in circuits.web.parsers.querystring), 108, 235, 278, 281
QueryStringToken (class in circuits.web.parsers.querystring), 108, 235, 278, 281
qvalue (circuits.web.headers.AcceptElement attribute), 119, 246, 272

## R

RangeUnsatisfiable, 117, 244, 270
read (class in circuits.io.events), 82, 179, 193
read (class in circuits.net.events), 91, 202, 208
read() (circuits.protocols.http.ResponseObject method), 98, 220, 224
read() (circuits.web.servers.StdinServer method), 123, 250, 282
ready (class in circuits.io.events), 82, 179, 194
ready (class in circuits.net.events), 91, 202, 208
ready() (circuits.net.sockets.UNIXClient method), 93, 205, 211
realm (circuits.web.main.Authentication attribute), 121, 248, 274
reason (circuits.web.wrappers.HTTPStatus attribute), 127, 254, 289
recv_body() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 279
recv_body_into() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 279
Redirect, 115, 242, 268
redirect (class in circuits.web.errors), 112, 239, 265
redirect() (circuits.web.controllers.BaseController method), 111, 237, 257
reduce_time_left() (circuits.core.events.generate_events method), 60, 142, 163
register() (circuits.core.BaseComponent method), 69, 152
register() (circuits.core.components.BaseComponent method), 55, 138, 158
registerChild() (circuits.core.Manager method), 73, 156
registerChild() (circuits.core.manager.Manager method), 63, 146, 166
registered (class in circuits.core.events), 58, 141, 162
registered() (circuits.app.Daemon method), 54, 133
registered() (circuits.app.daemon.Daemon method), 51, 131, 134
registerTask() (circuits.core.Manager method), 73, 156
registerTask() (circuits.core.manager.Manager method), 63, 146, 166
relative() (circuits.web.url.URL method), 126, 253, 286
remote (circuits.web.wrappers.Request attribute), 128, 255, 290
remote (class in circuits.node.events), 95, 212, 217
remove_move() (in module circuits.six), 129, 227
removeHandler() (circuits.core.Manager method), 73, 156
removeHandler() (circuits.core.manager.Manager method), 63, 145, 166

removeReader() (circuits.core.pollers.BasePoller method), 64, 147, 168
removeReader() (circuits.core.pollers.EPoll method), 65, 148, 168
removeReader() (circuits.core.pollers.KQueue method), 65, 148, 169
removeReader() (circuits.core.pollers.Poll method), 65, 148, 168
removeWriter() (circuits.core.pollers.BasePoller method), 64, 147, 168
removeWriter() (circuits.core.pollers.EPoll method), 65, 148, 168
removeWriter() (circuits.core.pollers.KQueue method), 65, 148, 169
removeWriter() (circuits.core.pollers.Poll method), 65, 148, 168
replace() (circuits.web.parsers.multipart.MultiDict method), 107, 233, 276, 280
reprhandler() (in module circuits.core.handlers), 61, 143, 164
request (class in circuits.protocols.http), 98, 220, 224
request (class in circuits.web.client), 110, 237, 257
request (class in circuits.web.events), 112, 239, 265
Request (class in circuits.web.wrappers), 127, 254, 289
request() (circuits.web.client.Client method), 110, 236, 256
request() (circuits.web.main.Authentication method), 121, 248, 274
request() (circuits.web.main.HelloWorld method), 121, 248, 274
request() (circuits.web.sessions.Sessions method), 123, 250, 283
RequestEntityTooLarge, 116, 242, 268
RequestTimeout, 116, 242, 268
RequestURITooLarge, 116, 243, 269
reraise() (in module circuits.six), 130, 228
reset() (circuits.core.Timer method), 72, 155
reset() (circuits.core.timers.Timer method), 66, 149, 169
resolve_methods() (in module circuits.web.dispatchers.dispatcher), 103, 229, 258, 261
resolve_path() (in module circuits.web.dispatchers.dispatcher), 103, 229, 258, 261
response (circuits.web.client.Client attribute), 110, 236, 256
response (class in circuits.protocols.http), 99, 221, 224
response (class in circuits.web.events), 113, 240, 266
Response (class in circuits.web.wrappers), 128, 255, 290
response() (circuits.web.wsgi.Application method), 129, 256, 291
ResponseObject (class in circuits.protocols.http), 98, 220, 224
resume() (circuits.core.helpers.FallBackGenerator method), 61, 144, 164
resume() (circuits.core.pollers.BasePoller method), 64, 147, 168
ReverseProxy (class in circuits.web.tools), 124, 251, 283
Root (class in circuits.web.main), 122, 248, 274
rpc (class in circuits.web.dispatchers.jsonrpc), 103, 229, 259, 262
rpc (class in circuits.web.dispatchers.xmlrpc), 104, 231, 260, 263
run() (circuits.core.Manager method), 73, 156
run() (circuits.core.manager.Manager method), 64, 147, 167
running (circuits.core.Manager attribute), 73, 157
running (circuits.core.manager.Manager attribute), 62, 145, 166

## S

safeimport() (in module circuits.core.utils), 66, 149, 169
sanitize() (circuits.web.errors.httperror method), 112, 238, 264
sanitize() (circuits.web.url.URL method), 126, 253, 286
save() (circuits.web.sessions.Sessions method), 123, 250, 283
save_as() (circuits.web.parsers.multipart.MultipartPart method), 108, 234, 277, 280
scheme (circuits.web.http.HTTP attribute), 121, 248, 274
scheme (circuits.web.wrappers.Request attribute), 128, 255, 290
script_name (circuits.web.wrappers.Request attribute), 128, 255, 290
secure (circuits.web.servers.BaseServer attribute), 122, 250, 282
secure (circuits.web.servers.StdinServer attribute), 123, 250, 282
secure (circuits.web.wsgi.Application attribute), 129, 256, 291
seek (class in circuits.io.events), 83, 180, 194
seek() (circuits.io.file.File method), 87, 184, 198
Select (class in circuits.core.pollers), 64, 147, 168
select_poller() (in module circuits.web.main), 122, 249, 274
send() (circuits.node.client.Client method), 94, 212, 216
send() (circuits.node.server.Server method), 97, 215, 219
send_all() (circuits.node.server.Server method), 98, 215, 219
send_to() (circuits.node.server.Server method), 98, 215, 219
Serial (class in circuits.io.serial), 88, 185, 199
serve_download() (circuits.web.controllers.BaseController method), 111, 237, 258
serve_download() (in module circuits.web.tools), 125, 252, 284

serve_file() (circuits.web.controllers.BaseController method), 111, 238, 258
serve_file() (in module circuits.web.tools), 125, 252, 284
server (circuits.web.wrappers.Request attribute), 128, 255, 290
Server (class in circuits.net.sockets), 92, 203, 210
Server (class in circuits.node.server), 97, 214, 218
Server (class in circuits.web.servers), 123, 250, 282
ServiceUnavailable, 116, 243, 269
Sessions (class in circuits.web.sessions), 123, 250, 283
setdefault() (circuits.web.headers.CaseInsensitiveDict method), 119, 246, 272
setValue() (circuits.core.values.Value method), 66, 150, 170
should_keep_alive() (circuits.web.parsers.http.HttpParser method), 106, 232, 275, 279
signal (class in circuits.core.events), 58, 141, 161
signal() (circuits.io.process.Process method), 87, 184, 199
Sleep (class in circuits.core.manager), 62, 145, 165
sleep() (in module circuits.core.manager), 62, 145, 165
socket_family (circuits.net.sockets.TCP6Client attribute), 93, 204, 210
socket_family (circuits.net.sockets.TCP6Server attribute), 93, 204, 210
socket_family (circuits.net.sockets.TCPClient attribute), 93, 204, 210
socket_family (circuits.net.sockets.TCPServer attribute), 93, 204, 210
socket_family (circuits.net.sockets.UDP6Server attribute), 93, 204, 210
socket_family (circuits.net.sockets.UDPServer attribute), 93, 205, 211
splitLines() (in module circuits.protocols.line), 101, 223, 227
start() (circuits.core.Manager method), 73, 157
start() (circuits.core.manager.Manager method), 64, 146, 167
start() (circuits.io.process.Process method), 87, 184, 199
started (class in circuits.core.events), 58, 140, 161
started (class in circuits.io.events), 84, 181, 195
Static (class in circuits.web.dispatchers.static), 104, 230, 260, 262
status (circuits.io.process.Process attribute), 87, 184, 199
status (circuits.web.wrappers.HTTPStatus attribute), 127, 254, 289
status (circuits.web.wrappers.Response attribute), 128, 255, 290
Status (class in circuits.web.wrappers), 128, 255, 290
stddev() (in module circuits.web.utils), 127, 254, 286
StdinServer (class in circuits.web.servers), 123, 250, 282
stop() (circuits.core.Event method), 70, 153
stop() (circuits.core.events.Event method), 57, 140, 161
stop() (circuits.core.Manager method), 73, 157
stop() (circuits.core.manager.Manager method), 64, 146, 167
stop() (circuits.io.process.Process method), 87, 185, 199
stopped (class in circuits.core.events), 58, 141, 161
stopped (class in circuits.io.events), 85, 182, 196
stream (circuits.web.wrappers.Response attribute), 128, 255, 290
stream (class in circuits.web.events), 114, 241, 267
success (circuits.core.Event attribute), 70, 153
success (circuits.core.events.Event attribute), 57, 140, 160
success (circuits.core.task attribute), 70, 153
success (circuits.core.workers.task attribute), 67, 150, 170
success (circuits.web.events.request attribute), 113, 240, 266
success (circuits.web.events.response attribute), 114, 241, 266
success (circuits.web.events.stream attribute), 115, 241, 267

## T

task (circuits.core.manager.Sleep attribute), 62, 145, 165
task (class in circuits.core), 70, 153
task (class in circuits.core.workers), 67, 150, 170
TCP6Client (class in circuits.net.sockets), 92, 204, 210
TCP6Server (class in circuits.net.sockets), 93, 204, 210
TCPClient (class in circuits.net.sockets), 93, 204, 210
TCPServer (class in circuits.net.sockets), 93, 204, 210
terminate (class in circuits.web.events), 115, 241, 267
tick() (circuits.core.Manager method), 73, 157
tick() (circuits.core.manager.Manager method), 64, 147, 167
time_left (circuits.core.events.generate_events attribute), 59, 142, 163
TimeoutError, 61, 74, 144, 157, 165
Timer (class in circuits.core), 71, 154
Timer (class in circuits.core.timers), 65, 148, 169
tob() (in module circuits.web.parsers.multipart), 108, 235, 277, 281
tokens() (circuits.web.parsers.querystring.QueryStringParser method), 108, 235, 278, 281
traceback (circuits.web.exceptions.HTTPException attribute), 117, 244, 269
translateHeaders() (circuits.web.wsgi.Application method), 129, 256, 291
tryimport() (in module circuits.tools), 102, 228

## U

u() (in module circuits.six), 130, 228
UDP6Client (in module circuits.net.sockets), 93, 204, 210
UDP6Server (class in circuits.net.sockets), 93, 204, 210
UDPClient (in module circuits.net.sockets), 93, 204, 210
UDPServer (class in circuits.net.sockets), 93, 204, 210
Unauthorized, 117, 244, 270
unauthorized (class in circuits.web.errors), 112, 239, 265

unescape() (circuits.web.url.URL method), 126, 253, 286
unicode() (circuits.web.url.URL method), 126, 253, 286
UnicodeError, 117, 244, 269
UNIXClient (class in circuits.net.sockets), 93, 205, 211
UNIXServer (class in circuits.net.sockets), 93, 205, 211
Unknown (class in circuits.core.handlers), 61, 143, 164
unmounted (class in circuits.io.events), 85, 182, 196
unpunycode() (circuits.web.url.URL method), 126, 253, 286
unreachable (class in circuits.net.events), 91, 203, 209
unregister() (circuits.core.BaseComponent method), 69, 152
unregister() (circuits.core.components.BaseComponent method), 55, 138, 158
unregister_pending (circuits.core.BaseComponent attribute), 69, 152
unregister_pending (circuits.core.components.BaseComponent attribute), 56, 138, 159
unregisterChild() (circuits.core.Manager method), 74, 157
unregisterChild() (circuits.core.manager.Manager method), 63, 146, 166
unregistered (class in circuits.core.events), 58, 141, 162
unregisterTask() (circuits.core.Manager method), 74, 157
unregisterTask() (circuits.core.manager.Manager method), 63, 146, 166
UnregistrableError, 61, 144, 165
UnsupportedMediaType, 116, 243, 269
update() (circuits.web.headers.CaseInsensitiveDict method), 119, 246, 272
uri (circuits.web.controllers.BaseController attribute), 111, 238, 258
URL (class in circuits.web.url), 125, 252, 285
users (circuits.web.main.Authentication attribute), 121, 248, 274
utf8() (circuits.web.url.URL method), 126, 253, 286

## V

validate_etags() (in module circuits.web.tools), 125, 252, 285
validate_since() (in module circuits.web.tools), 125, 252, 285
value (circuits.core.values.Value attribute), 66, 150, 170
value (circuits.web.parsers.multipart.MultipartPart attribute), 108, 234, 277, 280
Value (class in circuits.core.values), 66, 149, 170
variance() (in module circuits.web.utils), 127, 254, 286
VCS, **299**
verify_session() (in module circuits.web.sessions), 123, 250, 283
version (circuits.web.http.HTTP attribute), 121, 248, 274
VirtualHosts (class in circuits.web.dispatchers.virtualhosts), 104, 230, 260, 263

## W

wait() (circuits.core.Manager method), 74, 157
wait() (circuits.core.manager.Manager method), 63, 146, 166
wait() (circuits.io.process.Process method), 87, 185, 199
waitEvent() (circuits.core.Manager method), 74, 157
waitEvent() (circuits.core.manager.Manager method), 63, 146, 166
waitingHandlers (circuits.core.Event attribute), 70, 153
waitingHandlers (circuits.core.events.Event attribute), 57, 140, 160
walk() (in module circuits.tools), 102, 228
WebSocketClient (class in circuits.web.websockets.client), 109, 235, 287, 288
WebSocketCodec (class in circuits.protocols.websocket), 101, 223, 227
WebSocketsDispatcher (class in circuits.web.websockets.dispatcher), 109, 236, 287, 288
who() (in module circuits.web.sessions), 123, 250, 283
with_metaclass() (in module circuits.six), 130, 228
Worker (class in circuits.core), 70, 154
Worker (class in circuits.core.workers), 67, 150, 171
write (class in circuits.io.events), 86, 183, 197
write (class in circuits.net.events), 92, 203, 209
write() (circuits.io.file.File method), 87, 184, 198
write() (circuits.io.process.Process method), 88, 185, 199
write() (circuits.io.serial.Serial method), 88, 185, 199
write() (circuits.net.sockets.Client method), 92, 203, 209
write() (circuits.net.sockets.Server method), 92, 204, 210
write() (circuits.net.sockets.UDPServer method), 93, 205, 211
write() (circuits.web.client.Client method), 110, 236, 257
write() (circuits.web.servers.StdinServer method), 123, 250, 283
write_body() (circuits.web.parsers.multipart.MultipartPart method), 108, 234, 277, 280
write_header() (circuits.web.parsers.multipart.MultipartPart method), 108, 234, 277, 280
writepid (class in circuits.app.daemon), 52, 132, 136
writepid() (circuits.app.Daemon method), 54, 133
writepid() (circuits.app.daemon.Daemon method), 51, 131, 134

## X

XMLRPC (class in circuits.web.dispatchers.xmlrpc), 104, 231, 260, 263